Recap

Process data structures:

- User space
- PCB (process control block)

Multiprogramming:

- Scheduler, Process states/queues
- Context switch

Process API: fork, wait, exec, pipe, dup

xv6-riscv

L3: Limited Direct Execution

(Based on Chapter 6)

Spring 2023

Slides revised from
Matthew Tancreti
Iowa State University

How to run processes?

Direct Execution (without limits)

OS

1. Create entry for process
2. Allocate memory for program
3. Load program into memory
4. Set up stack (argc and argv parameters for main)
5. Clear registers
6. Execute **call** main( )

Process

1. Run main( )
2. Execute **return** from main

OS

1. Free memory of process
2. Remove from process list

What is wrong?

- Process has unrestricted access to memory and other resources.
- OS has no way to switch another process, must wait for program to finish.

How to restrict process' operations?

Restricting Access with Processor Mode

CPU has bit that indicates if in *user mode* or *kernel mode*

When in user mode, *"privileged" instructions* not allowed and memory boundaries are enforced

- Cannot read/write outside of address space bounds
- Cannot read/write I/O devices

When in kernel mode, all instructions are allowed

User processes only execute in user mode, the OS executes in kernel mode

Processor Mode

OS — Kernel Mode

1. Create entry for process
2. Allocate memory for program
3. Load program into memory
4. Set up stack (argc and argv parameters for main
5. Clear registers
6. Execute **call** main( )

enter user mode

Process — User Mode

1. Run main( )
2. Execute **return** from main

enter kernel mode

1. Free memory of process
2. Remove from process list

Process can't access out of its memory space and can't read/write I/O devices

System Call

Problem: How can a process perform privileged operations, for example read from a file? *System call!*

If no different modes, user could just make a functional call to "kernel" libraries.

A *system call* differs from a normal function call: it transfers control to the OS.

A user process initiates a system call by causing a *trap* (a software generated interrupt)

- For example, the RISC V instruction that causes a trap is *ecall*

System Call

OS | Kernel Mode

Hardware

Process | User Mode

Initialize process and **return-from-trap** into main( )

Return-from-trap changes mode to user

1. Run main( )
2. System call causes **trap**

Trap changes mode to kernel

1. Handle trap
   …
2. return-from-trap

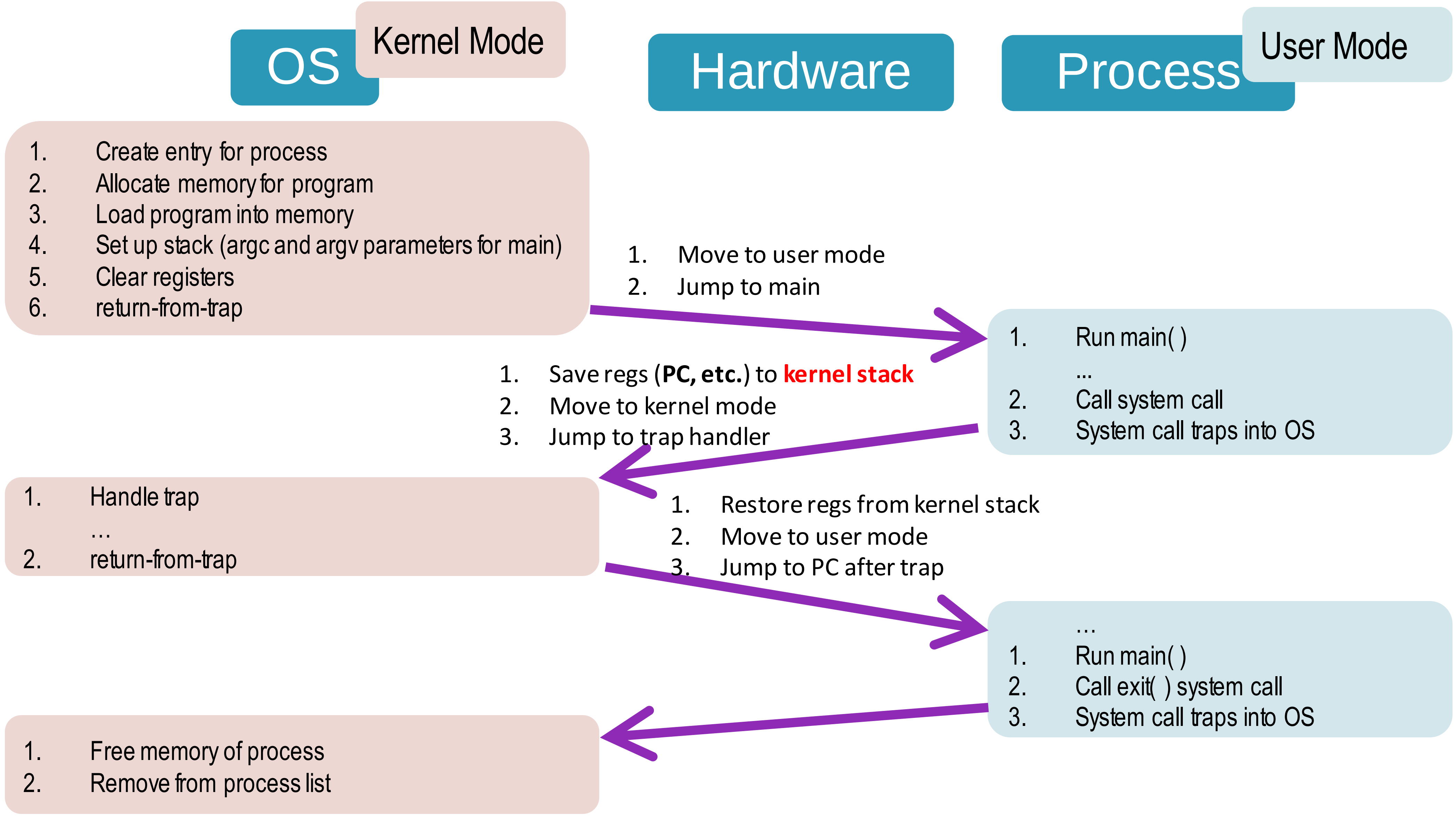
Return-from-trap changes mode to user

…
1. Call **exit**( ) system call
2. System call traps into OS

Trap changes mode to kernel

Free memory and remove process

System Call Details



Process Control Block (struct proc) in xv6

proc.h

```
// Saved registers for kernel context switches.
struct context {
  uint64 ra;
  uint64 sp;

  // callee-saved
  uint64 s0;
  uint64 s1;
  uint64 s2;
  uint64 s3;
  uint64 s4;
  uint64 s5;
  uint64 s6;
  uint64 s7;
  uint64 s8;
  uint64 s9;
  uint64 s10;
  uint64 s11;
};

enum procstate { UNUSED, USED, SLEEPING,
RUNNABLE, RUNNING, ZOMBIE };
```

```
// Per-process state
struct proc {
  struct spinlock lock;

  // p->lock must be held when using these:
  enum procstate state;        // Process state
  void *chan;                  // If non-zero, sleeping on channel
  int killed;                  // If non-zero, have been killed
  int xstate;                  // Exit status to be returned to parent's wait
  int pid;                     // Process ID

  // proc_tree_lock must be held when using this:
  struct proc *parent;         // Parent process

  // these are private to the process, so p->lock need not be held.
  uint64 kstack;               // Virtual address of kernel stack
  uint64 sz;                   // Size of process memory (bytes)
  pagetable_t pagetable;       // User page table
  struct trapframe *trapframe; // data page for trampoline.S
  struct context context;      // swtch() here to run process
  struct file *ofile[NOFILE];  // Open files
  struct inode *cwd;           // Current directory
  char name[16];               // Process name (debugging)
};
```

System Call Details: trap table

OS

Kernel Mode

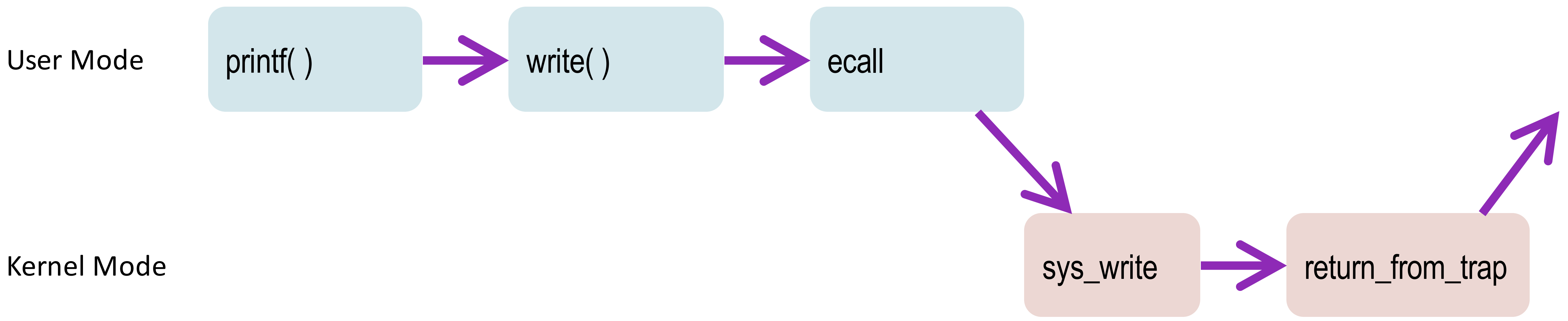
Hardware

Initialize trap table (map: interrupt number -> address of interrupt handler)

Remember address of syscall handler

System Call Example

Library functions like printf typically have some code that executes in user mode, it then makes one or more system calls.



How to switch between process?

More fundamentally, how the OS can regain the control after it starts a process?

A Cooperative Approach: Wait for System Calls

A well-behaving process makes a *yield* system call after it has executed for a while.

A process initializes an I/O operation by making a system call (e.g., *read, write*).

Once trap to OS: scheduler can switch process.

A Non-cooperative Approach: Timer Interrupt

OS Kernel Mode

Hardware

Initialize trap table (map: interrupt number ==> address of interrupt handler)

Remember address of syscall handler

Start interrupt timer

Start timer
Interrupt CPU in X ms

OS Regain Control via Timer Interrupt

OS

Kernel Mode

Hardware

Process

User Mode

Process A

… …

Timer interrupt

Save regs(A) -> kernel-stack(A)

Move to kernel mode

Jump to trap handler

Handle the trap

Call switch() routine:

Save regs(A) -> proc_t(A)

Restore regs(B) <- proc_t(B)

Switch to kernel-stack(B)

return-from-trap (into B)

Restore regs(B) <- kernel-stack(B)

Move to user mode

Jump to B's PC

Process B

… …

Process Control Block (struct proc) in xv6

proc.h

```
// Saved registers for kernel context switches.
struct context {
  uint64 ra;
  uint64 sp;

  // callee-saved
  uint64 s0;
  uint64 s1;
  uint64 s2;
  uint64 s3;
  uint64 s4;
  uint64 s5;
  uint64 s6;
  uint64 s7;
  uint64 s8;
  uint64 s9;
  uint64 s10;
  uint64 s11;
};

enum procstate { UNUSED, USED, SLEEPING,
RUNNABLE, RUNNING, ZOMBIE };
```

```
// Per-process state
struct proc {
  struct spinlock lock;

  // p->lock must be held when using these:
  enum procstate state;        // Process state
  void *chan;                  // If non-zero, sleeping on channel
  int killed;                  // If non-zero, have been killed
  int xstate;                  // Exit status to be returned to parent's wait
  int pid;                     // Process ID

  // proc_tree_lock must be held when using this:
  struct proc *parent;         // Parent process

  // these are private to the process, so p->lock need not be held.
  uint64 kstack;               // Virtual address of kernel stack
  uint64 sz;                   // Size of process memory (bytes)
  pagetable_t pagetable;       // User page table
  struct trapframe *trapframe; // data page for trampoline.S
  struct context context;      // swtch() here to run process
  struct file *ofile[NOFILE];  // Open files
  struct inode *cwd;           // Current directory
  char name[16];               // Process name (debugging)
};
```

swtch.S

```
# Context switch
#
#   void swtch(struct context *old, struct context *new);
#
# Save current registers in old. Load from new.


.globl swtch
swtch:
    sd ra, 0(a0)
    sd sp, 8(a0)
    sd s0, 16(a0)
    sd s1, 24(a0)
    sd s2, 32(a0)
    sd s3, 40(a0)
    sd s4, 48(a0)
    sd s5, 56(a0)
    sd s6, 64(a0)
    sd s7, 72(a0)
    sd s8, 80(a0)
    sd s9, 88(a0)
    sd s10, 96(a0)
    sd s11, 104(a0)
    ld ra, 0(a1)
    ld sp, 8(a1)
    ld s0, 16(a1)
    ld s1, 24(a1)
    ld s2, 32(a1)
    ld s3, 40(a1)
    ld s4, 48(a1)
    ld s5, 56(a1)
    ld s6, 64(a1)
    ld s7, 72(a1)
    ld s8, 80(a1)
    ld s9, 88(a1)
    ld s10, 96(a1)
    ld s11, 104(a1)

    ret
```

store registers to old context
sd = store doubleword command

load new context to registers
ld = load doubleword command

Limited Direct Execution

Time-sharing is the idea that multiple processes can run together on a single machine as though they are in sole control of the machine

How do processes share time?

multiprogramming – when a process waits for I/O, the OS can have another take over the CPU

multitasking – each process gets a time-slice, a time limit before the next process gets to execute on the CPU

How does the OS keep control?

Hardware provides *interrupts*, *kernel mode* and *user mode*

System calls (cooperative) and interrupts (non-cooperative) enable OS to regain control after starting a process.