Process management 2

CS503: Operating systems, Spring 2019

Pedro Fonseca
Department of Computer Science
Purdue University

Edited by Byoungyoung Lee and Pedro Fonseca

Admin

- One more TA: Hafiz Kamran Khalil :)
- Next lab is scheduled to be released next week

Previous lecture

- Process abstraction
  - Heavy process vs. light process
- Process structures kept by the kernel
  - Process table
- Scheduling and scheduling policies
- Context switch
  - O(1) only for picking the first process
    - But insertion takes more time see the implementation

Goals for today

- Continue understanding process management:
  - How to manage concurrent executions in an operating system?
  - What structures the OS needs to keep track off?
  - What is and how is a context switch implemented?
- What are the main process management services?

How are arguments passed in Xinu?

- Objdump -S xinu.elf

How are arguments passed in Xinu?

Arg

```
113dba:        c7 00 00 00 00 00        movl   $0x0,(%eax)
113dc0:        83 ec 0c                 sub    $0xc,%esp
113dc3:        6a 02                    push   $0x2
113dc5:        e8 1a 0d ff ff           call   104ae4 <resched_cntl>
113dca:        83 c4 10                 add    $0x10,%esp
113dcd:        83 ec 0c                 sub    $0xc,%esp
113dd0:        ff 75 f4                 pushl  -0xc(%ebp)
```

```
/*------------------------------------------------------------------------
 *  resched_cntl  -  Control whether rescheduling is deferred or allowed
 *------------------------------------------------------------------------
 */
status  resched_cntl(          /* Assumes interrupts are disabled      */
          int32 defer          /* Either DEFER_START or DEFER_STOP     */
        )
{
```

Assembly background

- Assembly:
  - one statement -> one machine instruction
  - assembler directives and labels
  - instruction format is restricted by architecture
- AT&T assembly syntax format:
  - [instruction mnemonic] [src] [destination]
  - Intel manuals use another format (i.e., Intel assembly syntax)
- Operands:
  - Register value: *%rax*
  - Memory location pointed by a register value: *(%rax)*
  - Memory location of a register value + offset: *12(%ebp)*
  - Some instructions change implicit registers (e.g., flags is changed by CMP and SUB)
  - Some instructions read implicit registers (e.g., JMP reads flags)

More info: https://en.wikibooks.org/wiki/X86_Assembly/GAS_Syntax

Context switch

- Restoring the CPU changes the CPU state itself
- Challenge is to make it atomic and avoid loss of information
- Particularly challenging is the PC register
    - It cannot even be read explicitly in x86

Context switch code

- `ctxsw()` in `system/ctxsw.S`

```
/*------------------------------------------------------------------------
 * ctxsw -  X86 context switch; the call is ctxsw(&old_sp, &new_sp)
 *------------------------------------------------------------------------
 */
ctxsw:
        pushl   %ebp            /* Push ebp onto stack          */
        movl    %esp,%ebp       /* Record current SP in ebp     */
        pushfl                  /* Push flags onto the stack    */
        pushal                  /* Push general regs. on stack */

                                /* Save old segment registers here, if multiple allowed */

        movl    8(%ebp),%eax    /* Get mem location in which to    */
                                /*   save the old process's SP     */
        movl    %esp,(%eax)     /* Save old process's SP       */
        movl    12(%ebp),%eax   /* Get location from which to */
                                /*   restore new process's SP */

                                /* The next instruction switches from the old process's   */
                                /*   stack to the new process's stack.             */

        movl    (%eax),%esp     /* Pop up new process's SP    */
        popal                   /* Restore general registers  */
        popfl                   /* Restore interrupt mask  */
        ret                     /* Return to new process   */
```

Why 3 pushes and only 2 pops?

(Buggy code in previous lecture)

Context switch code

- `ctxsw()` in `system/ctxsw.S`

```
ctxsw:
                pushl   %ebp            /* Push ebp onto stack          */
                movl    %esp,%ebp       /* Record current SP in ebp     */
                pushfl                  /* Push flags onto the stack    */
                pushal                  /* Push general regs. on stack  */

                /* Save old segment registers here, if multiple allowed */

                movl    8(%ebp),%eax    /* Get mem location in which to */
                                        /*   save the old process's SP  */
                movl    %esp,(%eax)     /* Save old process's SP        */
                movl    12(%ebp),%eax   /* Get location from which to   */
                                        /*   restore new process's SP   */

                /* The next instruction switches from the old process's */
                /*   stack to the new process's stack.                  */

                movl    (%eax),%esp     /* Pop up new process's SP      */

                /* Restore new seg. registers here, if multiple allowed */

                popal                   /* Restore general registers    */
                movl    4(%esp),%ebp    /* Pick up ebp before restoring */
                                        /*   interrupts                 */
                popfl                   /* Restore interrupt mask       */
                add     $4,%esp         /* Skip saved value of ebp      */
                ret                     /* Return to new process        */
```

Book and lab code also include two important instructions

Context switch

- Restoring the CPU changes the CPU state itself
- Challenge is to make it atomic and avoid loss of information
- Particularly challenging is the PC register
  - It cannot even be read explicitly in x86
  - *The trick is to not store the current PC value; instead we store the location where the ctxsw() should return*
  - *Using the **stack** and the **call**/**ret** instruction semantics*

Evolution of the x86 context switch in Linux

- Many ways to do the context switch in x86!
- Linux when through a long evolution
- HW support for context switching
- Multiprocessor challenges:
    - Migration of processes between CPUs
    - Each processor has a dedicated interrupt controller (APIC)

More info at:
http://www.maizure.org/projects/evolution_x86_context_switch_linux/

Hardware context switching

- Also known as “Hardware Task Switching” in CPU manuals
- May only handle part of the process state (e.g,. exclude FPU/MMX and SSE state)
- In x86: one mechanism relies on the task register (TR) and a far CALL or JMP to load a task state segment (TSS)
- Hardware context switching (vs. software context switching)
  - Exact mechanisms is very hardware specific
  - May perform unnecessary checks (e.g., segmentation)
  - In practice slow

Question #1

- Our invariant says that at any time, a process must be executing
- Context switch code moves from one process to another
- **Q**: which process executes the context switch code?

Question #2

- At any time, CPU should execute something
- **Q:** If all user processes may be idle (e.g., applications all wait for input), which process should OS executes?

Null process

- Does not compute anything useful
- Is present merely to ensure that at least one process remains ready at all times
- Simplifies scheduling (no special cases)

Code for a null process

- Easiest way to code a null process:

```
while (1);
    /* Do nothing */
```

- May not be optimal. **Q:** Why?
- Fetch-execute takes bus cycles
- **Q:** How to optimize it?
- Two possible ways:
  - Some processors offer a special pause instruction that stops the processor until an interrupt occurs
  - Instruction cache can eliminate bus accesses

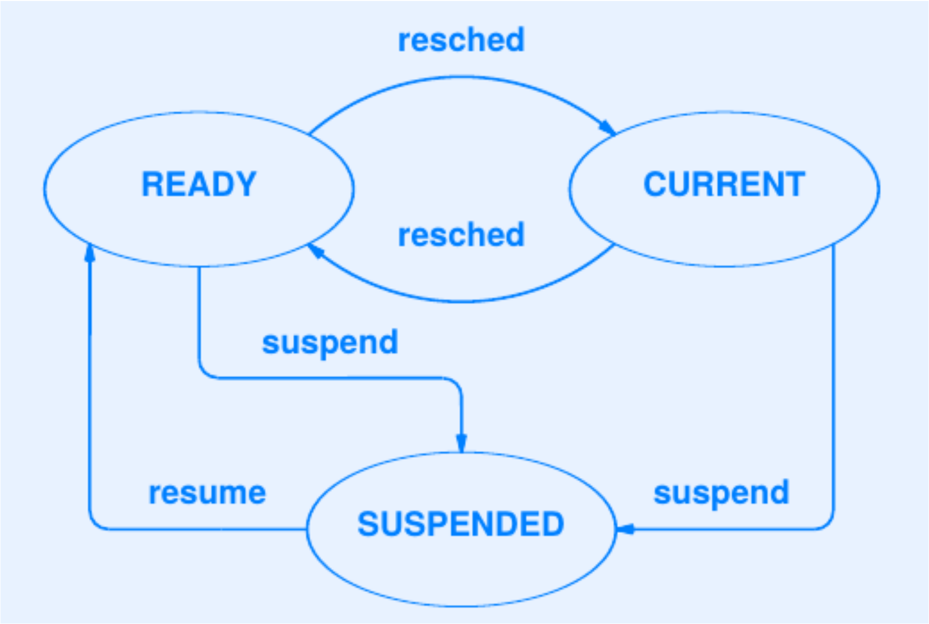
Process manipulation

- Need to invent ways to control processes
- Example operations:
  - Suspension
  - Resumption
  - Creation
  - Termination
- Recall: state variable in process table store the status

Process suspension

- Temporarily “stop” a process
- Prohibit it from using the processor
- To allow later resumption:
  - Process table entry retained
  - Complete state of computation saved
- OS sets process table entry to indicate process is suspended

State transition for suspension and resumption



- Either current or ready process can be suspended
- Only a suspended process can be resumed
- System calls can suspend or resume

Suspension code

- suspend() function

Process resumption

- Resume the execution of previously suspended process
- Method:
  - Make process eligible for processor
  - Re-establish scheduling invariant
- Note: resumption does not guarantee instantaneous execution

Resumption code

- resume() function

Function to make the process ready

- ready() function
- Make a process eligible for execution

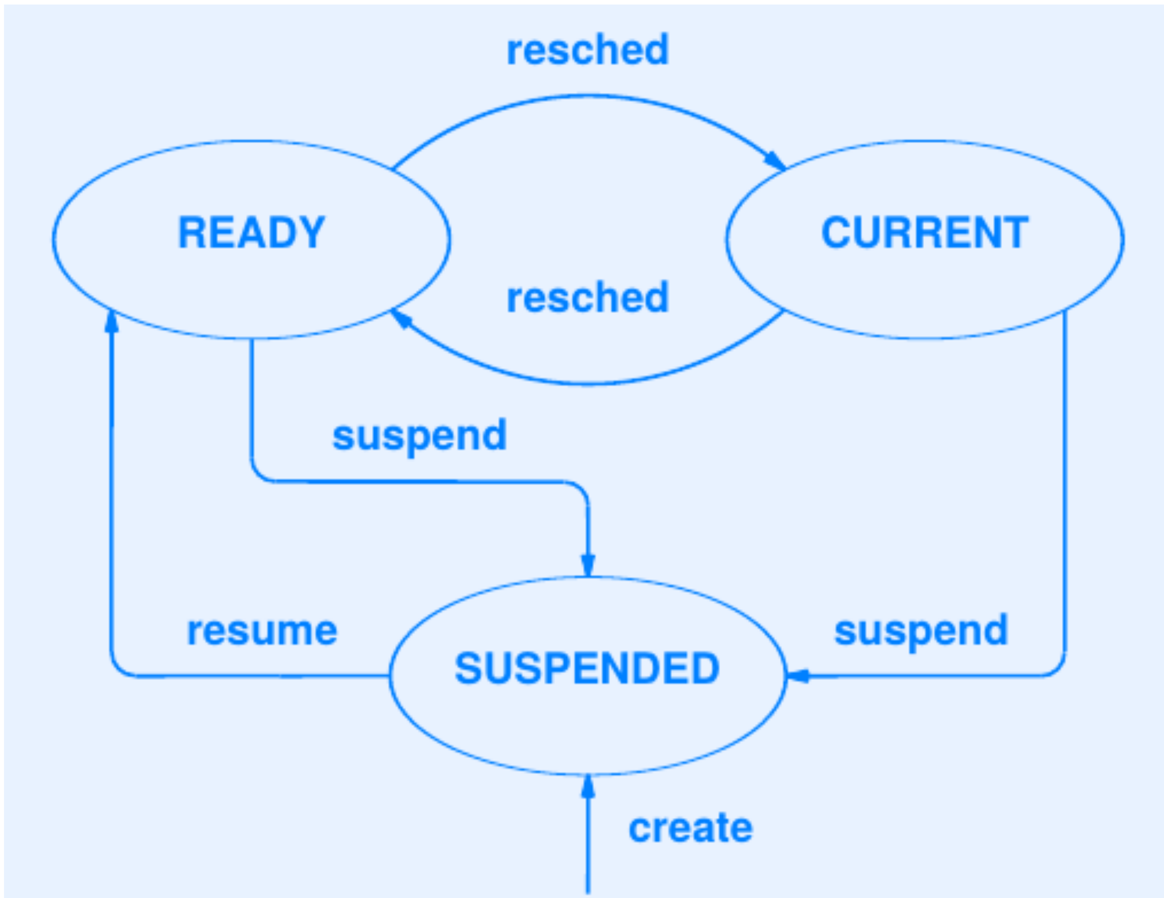
Process termination

- Final and permanent
- Record of a process is removed
- Process table entry becomes available for reuse
- Known as process exit if initiated by the thread itself
- We will see more about termination later

Process creation

- Processes are dynamic — process creation refers to starting a new process
- Implemented by **create** function in Xinu
- Method:
  - Find free entry in process table
  - Fill in entry
  - Place new process in suspended state
- Needs to select a new pid

State transition for additional process management functions



Note that both current and ready processes can be suspended

Summary

- Context switch
- Null process
- Process management functions:
    - Suspension
    - Resumption
    - Creation
    - Termination