Process management

CS503: Operating systems, Spring 2019

Pedro Fonseca
Department of Computer Science
Purdue University

Admin

- Labs due next Monday

Previous lecture

- Memory system
  - Endianness, caches
- Bus operations
- Calling conventions
- I/O communication
  - Interrupts
  - DMA

Previous lecture

- DMA:
  - Advantages:
    - Large transfers are more efficient
    - CPU and device *can* work in parallel
  - Disadvantages:
    - Higher per transfer overhead:
      - Need to configure the DMA controller
      - Only compensates for large transfers
    - Cache coherence problems

Goals for today

- 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?

Terminology

- The term **process management** has been used for decades to encompass the part of an operating system that manages concurrent execution, including both processes and the threads within them
- The term **thread management** is newer, but sometimes leads to confusion because it appears to exclude processes

Revisit: concurrent processing

- Unit of computation
- Abstraction of a processor
  - Known only to operating systems
  - Not known by hardware

A fundamental principle

- All computations must be done by a process
  - No execution by the OS itself
  - No execution outside of a process
- Key consequence:
  - At any time a process (or a thread) must be running
  - Operating system cannot stop running a process unless it switches to another process

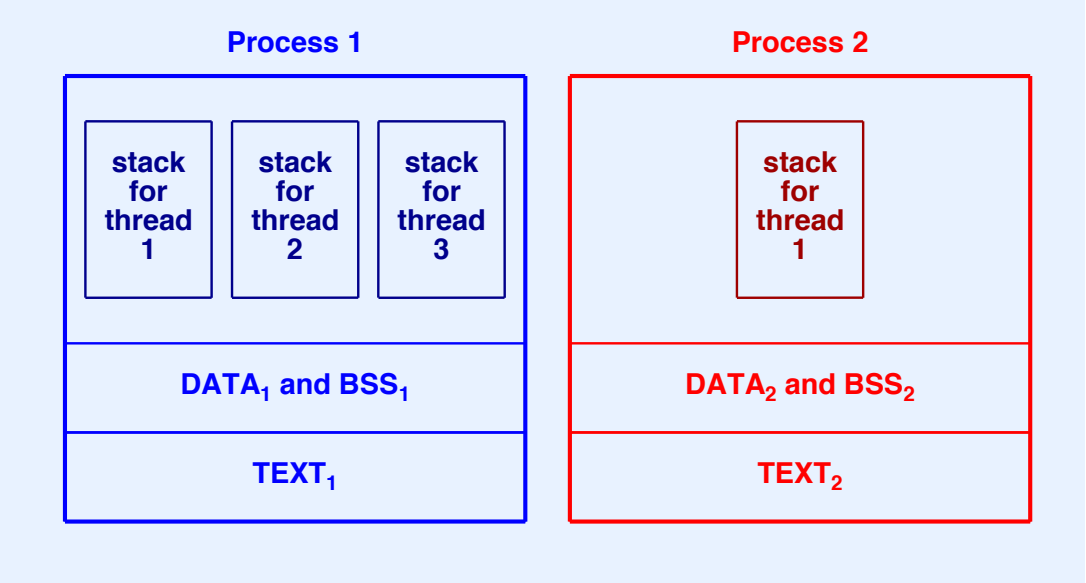
Heavyweight Process (or Process)

- Pioneered in Mach and adopted by Linux
- Address space in which multiple threads can execute
- One data segment per process
- One BSS segment per process
- Multiple threads per process

Lightweight Process (or Thread)

- Single “execution”
- Can share data (data and BSS segments) with other threads
- Must have a private stack segment
  - To keep track of execution contexts
  - Storing local variables, call stacks, etc.

Illustration of two heavyweight processes and their threads



- Threads within a process share text, data, and bss
- No sharing between Processes
- Threads within a Process cannot share stack

Maintaining processes

- Process
  - OS abstraction
  - Unknown to hardware
  - Created dynamically
- Information kept by OS
- OS stores information in a central data structure
  - Called process table
  - Part of OS address space

Information kept in a process table

- For each process:
  - Unique process identifier
  - Owner (a user)
  - Scheduling priority
  - Location of code and data (stack)
  - Process status (running, ready, wait, etc.)
- Q: How about **current program counter** and **current register** values?

Information kept in a process table

- For each thread (if used):
  - Owning process
  - Thread's scheduling priority
  - Location of stack
  - Status of computation

XINU process model

- Simple(st) scheme
- Single-user system (no ownership)
- One global context
- One global address space
- No boundary between OS and applications
- Note: all Xinu processes can share data

Xinu's process table

- **prstate**: The current status of the process (e.g., whether the prstate process is currently executing or waiting)
- **prprio**: The scheduling priority of the process
- **prstkptr**: The saved value of the process stack pointer when the process is not executing
- **prstkbase**: The address of the base of the proces's stack
- **prstklen**: A limit on the maximum size that the process's stack can grpow
- **prname**: A name assigned to the process that humans use to identify the process's purpose

Process states

- Used by OS to manage processes
- Set by OS whenever process changes status (e.g., waits for IO)
- Small enum value (integer) stored in the process table
- Tested by OS to determine:
  - Whether a requested operation is valid
  - The meaning of an operation

Process states

- Specified by OS designer
- One “state”” assigned per activity
- Values are updated in a process table when activity changes
- Example values:
  - **Current**: process is currently executing
  - **Ready**: process is ready to execute
  - **Waiting**: process is waiting on semaphore
  - **Receiving**: process is waiting to receive a message
  - **Sleeping**: process is delayed for specified time
  - **Suspended**: process is not permitted to execute

Definition of Xinu process state constants

```
// Process state constants

#define PR_FREE     0   /* Process table entry is unused    */
#define PR_CURR     1   /* Process is currently running     */
#define PR_READY    2   /* Process is on ready queue        */
#define PR_RECV     3   /* Process waiting for message      */
#define PR_SLEEP    4   /* Process is sleeping          */
#define PR_SUSP     5   /* Process is suspended         */
#define PR_WAIT     6   /* Process is on semaphore queue    */
#define PR_RECTIM   7   /* Process is receiving with timeout
*/
```

- States are defined as needed when system is constructed
- We will understand the process of each state as we consider the system design

Scheduling and context switching

Scheduling

- Fundamental part of process management
- Performed by the OS
- Three steps:
  - Examine the processes that are eligible for execution
  - Select a process to run
  - Switch the processor to the selected process

Implementation of scheduling

- The OS needs a **scheduling policy** that specifies which process to select
- We must then build a scheduling function that:
  - Selects a process according to the policy
  - Update the process table for the current and selected processes
  - Perform a **context switch:**
    - Switching from current process to the selected process

Scheduling policy

- Determines which process is selected for execution
- Typically one of the goals is fairness
- May depend on:
  - User
  - How many processes a user owns
  - Time a given process has been waiting to run
  - Priority of the process
- Note: both hierarchical or flat scheduling can be used

Example of scheduling policy in Xinu

- Each process assigned a priority:
  - Non-negative integer value
  - Initialized when process created
  - Can be changed at any time
- Scheduler always chooses to run an eligible process that has highest priority
- Policy is implemented by a system-wide invariant

The Xinu scheduling invariant

- “At any time, the processor must be executing a highest priority eligible process. Among processes with equal priority, scheduling is round robin.”
- Invariant must be enforced whenever:
  - The set of eligible processes changes
  - The priority of any eligible process changes
- Such changes only happen during:
  - A system call
  - An interrupt

Implementation of scheduling

- Process is eligible if state is **ready** or **current**
- To avoid searching process table during scheduling
  - Keep ready processes on linked list called **ready list**
  - Order the ready list by process priority
  - Selection of highest-priority process can be performed in a constant time (i.e., O(1))

High-speed scheduling decision

- Compare priority of **current** process to priority of first process on **ready list**
  - If current process has a higher priority, do nothing
  - Otherwise, extract the first process from the ready list and perform a context switch to switch the processor to the next process

Deferred rescheduling

- Delays enforcement of scheduling invariant
- Prevents rescheduling temporarily
  - A call to **resched_cntl(DEFER_START)** suspends rescheduling
  - A call to **resched_cntl(DEFER_STOP)** resumes normal scheduling
- Main purpose: allow device driver to make multiple processes ready before allowing any of them to run
- We will see an example later

Xinu scheduler details

- Before calling the scheduler:
  - Global variable **currpid** gives ID of process that is executing
  - **proctab[currpid].prstate** must be set to the desired next state for the current process
  - Q: How do you reference **currpid** if running on multi-core CPUs?
- If current process remains eligible and has highest priority, scheduler does nothing
  - Just returns and keeps executing the current process
- Otherwise, scheduler moves current process to the specified state and runs the highest priority **ready** process

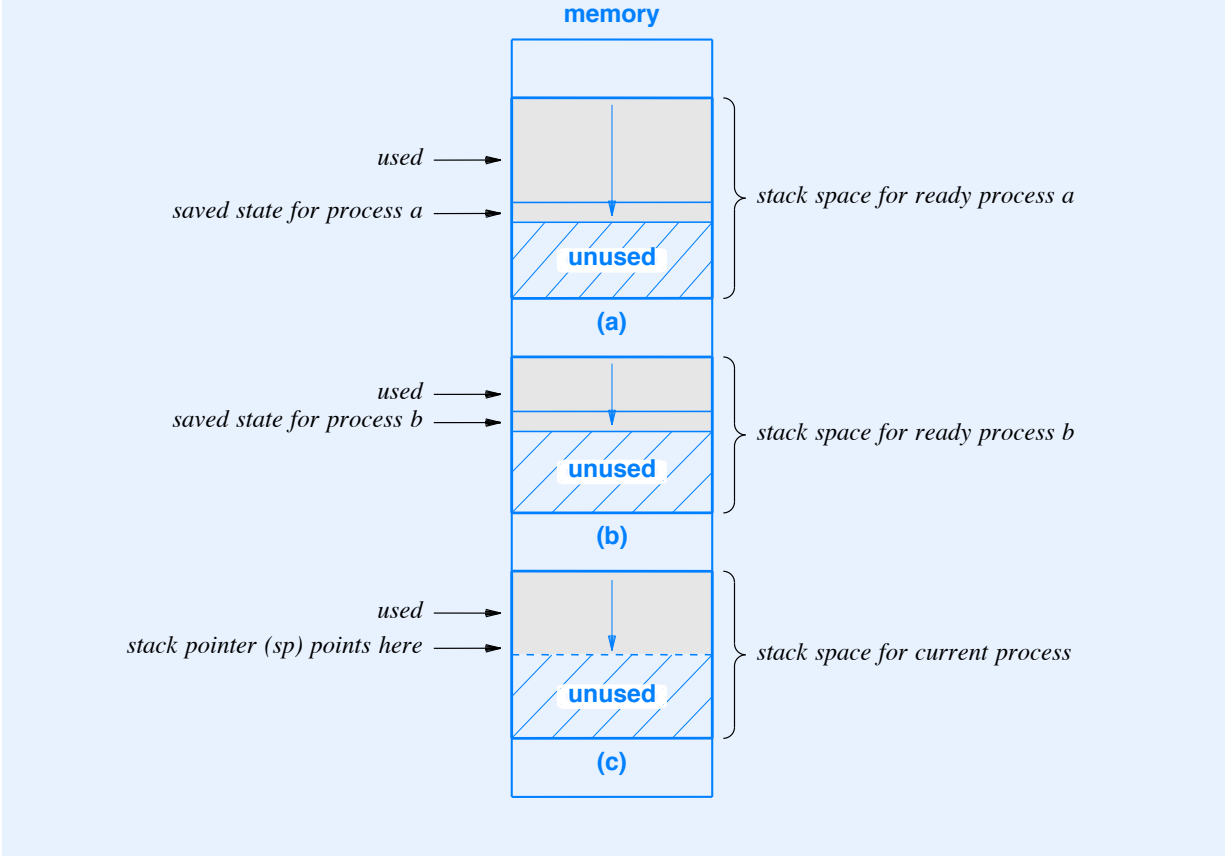
Xinu scheduler details

- What if several processes have the same priority and are the highest priority processes?
- When inserting a process on the ready list, places the process behind other processes with the same priority
  - Round-robin scheduling
- If scheduler switches context, first process on ready list is selected
- Note: scheduler switches context if the first process on the ready list has priority **equal** to the current process

Example scheduler code

- **resched()** and **resched_cntl()**
  - in **“system/resched.c”**

Illustration of states saved on process stack

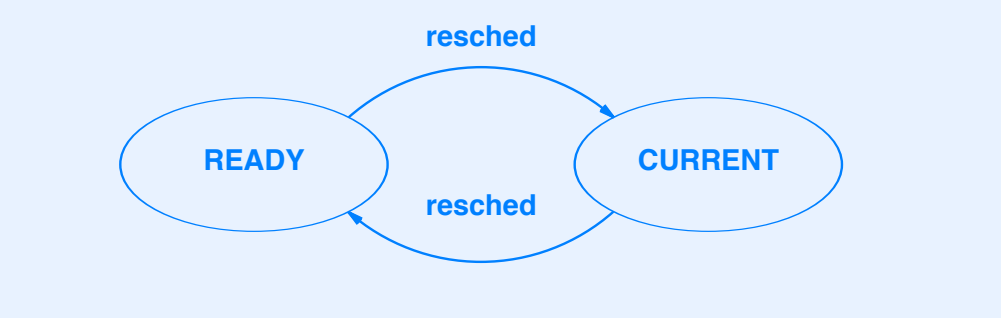


- Stack of each process contains the saved state

Process state transition

- Recall each process has a state
- State determines:
  - Whether an operation is valid
  - Semantics of each operation
- Transition diagram documents valid operations

Example: Transition between current and ready state



- **resched()** can move a process in either direction

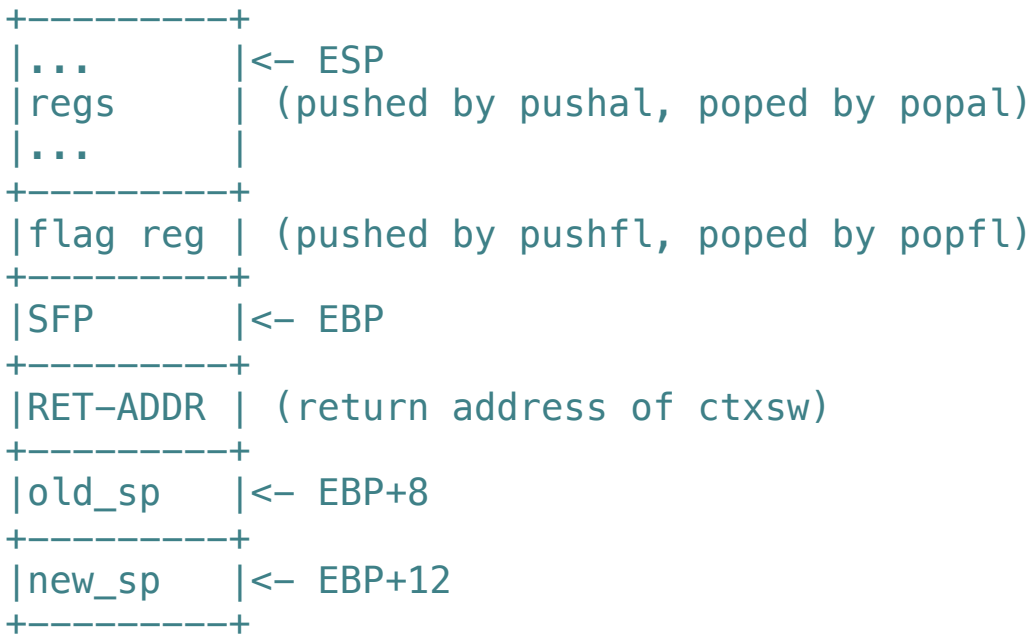
Context switch

- Component of process manager
- Low level (must manipulate underlying hardware)
- Usually written in assembly language
- Called by scheduler
- Moves processor from one process to another

Context switch operation

- Given a new process **N** and old process **O:**
- Save copy of all information pertinent to O on process O's stack or the process table entry for process O
  - Contents of HW registers
  - PC
  - Privilege level and hardware status
  - Memory map and address space
- Load saved information for N
- Resume execution of N

Context switch stack



low address values

high address values

(note diagram is in inverted order)

Context switch code (fixed)

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

```
10 ctxsw:
11                 pushl   %ebp            /* Push ebp onto stack          */
12                 movl    %esp,%ebp       /* Record current SP in ebp     */
13                 pushfl                  /* Push flags onto the stack    */
14                 pushal                  /* Push general regs. on stack  */
15
16                 /* Save old segment registers here, if multiple allowed */
17
18                 movl    8(%ebp),%eax    /* Get mem location in which to */
19                                         /*   save the old process's SP  */
20                 movl    %esp,(%eax)     /* Save old process's SP        */
21                 movl    12(%ebp),%eax   /* Get location from which to   */
22                                         /*   restore new process's SP   */
23
24                 /* The next instruction switches from the old process's */
25                 /*   stack to the new process's stack.                  */
26
27                 movl    (%eax),%esp     /* Pop up new process's SP      */
28
29                 /* Restore new seg. registers here, if multiple allowed */
30
31                 popal                   /* Restore general registers    */
32                 movl    4(%esp),%ebp    /* Pick up ebp before restoring */
33                                         /*   interrupts                 */
34                 popfl                   /* Restore interrupt mask       */
35                 add     $4,%esp         /* Skip saved value of ebp      */
36                 ret                     /* Return to new process        */
```

Book and lab code also include two important instructions

Summary

- Process management
- Process structures
- Scheduling and scheduling policies
- Context switch