Operating Systems

Context Switching
(and Scheduling)

1

Context Switching Implementation

- Registers can't be manipulated directly in C, so *resched* calls an assembly language function called *cxtsw*
  - This function is machine dependent
- The last step involves the program counter as Xinu must restore all other elements of program state before jumping to the new process
- Some architectures provide single instructions for (atomically) saving and restoring registers
  - The RISC model often requires a series of instructions – saving each register explicitly
  - ARM has push and pop macros

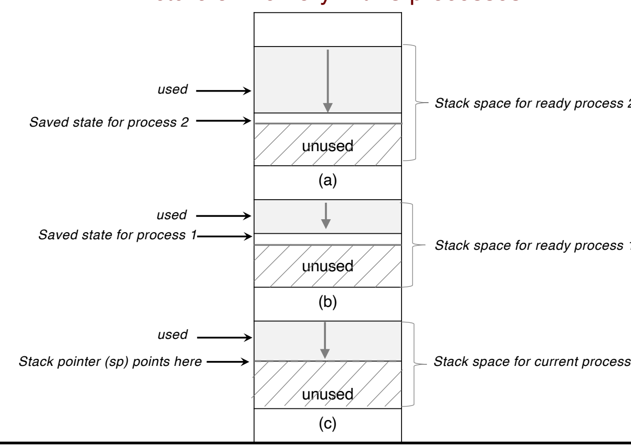
2

Saving State in Memory

- We have said that elements of state need to be stored on a per-process basis
- A process table entry is per-process but each process also has a stack
  - In use by the currently executing process
  - When a function is called, the executing process makes space on the stack for local variables and arguments
  - When it returns, they are popped off
  - Per-process even if processes are otherwise identical
- Xinu processes execute in this way already, and the call to *ctxsw* can use the same mechanism

3

Picture of Memory with 3 processes



4

ctxsw

- Xinu saves the process's register state on the process's stack
- Written in assembly
  - Rewritten for each architecture
- Thus *ctxsw*:
  - Executes instructions that push the register contents onto the stack of the running process
  - Saves the stack pointer in the process table entry for the current process, and loads the SP of the next process
  - Executes instructions to reload the processor registers from the values previously saved for the next process
  - Jumps to the location in the new process at which execution should resume

Context switch on ARM

- ARM has `push` and `pop` macros to copy multiple registers to the stack
  - On ARM (and MIPS) each instruction can store only one register; N instructions for N registers
- A "coprocessor" stores the status register, instruction `mrs` copies it to a general-purpose register
- r0 – r3 are caller-save, and ctxsw only uses 2 arguments so r2 and r3 are available
  - No need to save an extra register as on x86

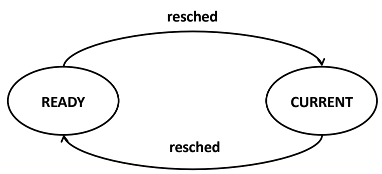

```
/* ctxsw.S - ctxsw (for ARM) */

        .text
        .globl  ctxsw

/*------------------------------------------------------------------------
 * ctxsw -  ARM context switch; the call is ctxsw(&old_sp, &new_sp)
 *------------------------------------------------------------------------
 */

ctxsw:
        push    {r0-r11, lr}            /* Push regs 0 - 11 and lr      */
        push    {lr}                    /* Push return address          */
        mrs     r2, cpsr                /* Obtain status from coprocess.*/
        push    {r2}                    /*   and push onto stack        */
        str     sp, [r0]                /* Save old process's SP        */
        ldr     sp, [r1]                /* Pick up new process's SP     */
        pop     {r0}                    /* Use status as argument and   */
        bl      restore                 /*   call restore to restore it */
        pop     {lr}                    /* Pick up the return address   */
        pop     {r0-r12}                /* Restore other registers      */
        mov     pc, r12                 /* Return to the new process    */
```


Switching and the Program Counter

- What we can observe is that a process calling resched and ctxsw will be suspended at that point – in ctxsw
- Saving and restoring the PC is a special case
  - It is changing as the instructions to save the registers are executed
  - It can specifically calculated
- When a process restarts it should begin executing in *resched* immediately following the call to *ctxsw*
- All processes call *resched* and it calls *ctxsw* so at context switch time, they are all frozen in the same place
  - But the stack above it is distinct

Concurrent execution

- The concurrent execution model means that a process must always be executing
  - and eventually execute the scheduler
- The scheduler's only function is to switch the process from one function to another



The Null process

- There are times when no regular process is ready to execute
  - blocked on a semaphore, waiting for I/O, etc.
- The scheduler needs a process to execute!
- So at least one process must remain to be switched to
- This is the Null process
  - PID 0
  - Priority 0
  - Infinite loop

Making a Process Ready

- When resched moves the current process onto the ready list, it does it directly
- Making a process ready is a frequent activity so there is a function
- Args: PID and a boolean to indicate whether resched should be called
  - RESCHED_{YES/NO}
- The scheduling policy says that the highest priority process is executing at any time
  - Scheduling invariant: a function assumes that the highest priority ready process was executing when it was called and must insure that is also true when it returns
  - If the function alters process state, then it must insure the invariant
- *ready* is an exception in that it may move multiple processes to the ready queue at once
  - Thus, RESCHED_NO


```
/* ready.c - ready */

#include <xinu.h>

qid16   readylist;          /* index of ready list    */

/*------------------------------------------------------------------------
 *  ready  -  Make a process eligible for CPU service
 *------------------------------------------------------------------------
 */
status  ready(
      pid32     pid,        /* ID of process to make ready */
      bool8     resch       /* reschedule afterward?   */
    )
{
    register struct procent *prptr;

    if (isbadpid(pid)) {
        return(SYSERR);
    }

    /* Set process state to indicate ready and add to ready list */

    prptr = &proctab[pid];
    prptr->prstate = PR_READY;
    insert(pid, readylist, prptr->prprio);
```

```
if (resch == RESCHED_YES) {
        resched();
    }
    return OK;
}
```

13

Deferred Rescheduling

- Resched uses Defer.ndefers to determine if rescheduling is deferred
  - sched_cntl(DEFER_START); /* defers, and */
  - sched_cntl(DEFER_STOP); /* ends deferral */
- It is a counter as multiple processes may request deferral
- When it is 0, then no process has requested deferral
- The Defer.attempt flag indicates whether it was attempted so that resched can be called

14

```
/* sched.h */

/* Constants and variables related to deferred rescheduling */

#define DEFER_START 1   /* start deferred rescheduling    */
#define DEFER_STOP  2   /* stop  deferred rescheduling    */

/* Structure that collects items related to deferred rescheduling  */

struct defer   {
    int32   ndefers;    /* number of outstanding defers   */
    bool8   attempt;    /* was resched called during the  */
                        /* deferral period?               */
};

extern  struct  defer   Defer;
```

15

```
/* sched_cntl.c - sched_cntl */

#include <xinu.h>

struct  defer   Defer;

/*------------------------------------------------------------------------
 *  sched_cntl  -  control whether rescheduling is deferred or allowed
 *------------------------------------------------------------------------
 */
status  sched_cntl(    /* assumes interrupts are disabled    */
      int32 def     /* either DEFER_START or DEFER_STOP   */
    )
{
switch (def) {

        /* Process request to defer:             */
        /*  1) Increment count of outstanding deferral requests */
        /*  2) If this is the start of deferral, initialize Boolean */
        /*     to indicate whether resched has been called  */
```

16

```
case DEFER_START:
        if (Defer.ndefers++ == 0) {   /* increment deferrals */
            Defer.attempt = FALSE; /* no attempts so far */
        }
        return OK;

        /* Process request to stop deferring:       */
        /*  1) Decrement count of outstanding deferral requests */
        /*  2) If last deferral ends, make up for any calls to  */
        /*     resched that were missed during the deferral */

        case DEFER_STOP:
        if (Defer.ndefers <= 0) { /* none outstanding */
            return SYSERR;
        }
        if (--Defer.ndefers == 0) {   /* end deferral period */
            if (Defer.attempt) {   /* resched was called */
                resched(); /*   during deferral   */
            }
        }
        return OK;

        default:
        return SYSERR;
    }
}
```


Context Switch on x86

- Must save registers for the old process on its stack
- x86 has a dedicated instruction to do this – `pushal`
  – and `popal` does the inverse
- First save EBX as it is needed to access the arguments
- Next, processor flags and general purpose registers
- Save the old process's stack pointer in the location pointed to by argument 1
- Load the new process's stack pointer in the location pointed to by argument 2


```
/* ctxsw.S - ctxsw (for x86) */

                .text
                .globl  ctxsw

/*------------------------------------------------------------------------
 * 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      */

/* 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        */
```