ToyOS: Hardware

ToyOS-1: no IO
ToyOS-2: Wait/Wakeup on PCB queue
Toy OS-3: IO: synchronous, no interrupt, no dma
Toy OS-4: IO: synchronous, interrupt, dma
Toy OS-5: IO: asynchronous, interrupt, dma
Overview

- **Goal:** to make concrete the essence of how an OS shares computer hardware securely among user processes.

- **Hardware:** processor, memory, timer, io-adaptor + device

- **Kernel + User processes**
- **Images initially in memory; no process creation**
- **Pseudo machine code**
- **Timer interrupt → context switch**
- **Address-space protection**
- **No filesystem, no ...**

- **ToyOS versions**
  - 1: no io
  - 2: wait/wakeup process synchronization
  - 3: io synchronous, no intrpts or DMA
  - 4: io synchronous, intrpts, DMA
  - 5: io asynchronous, intrpts, DMA
Processor (aka CPU) – 1

- State
  - general purpose regs (gpr)
  - instruction pointer (ip)
  - stack pointer (sp)
  - high-/low-address (hi, lo)
  - processor status (ps): intrpts on/off, mode kernel/user, ...

- Instructions
  - move, arith, logic, io, stack, control (jump, call, intrpt)
  - some are privileged: user-mode execution → exception

- Stack
  - push reg: mem[sp] ← reg; sp−−
    // or sp++
  - pop reg: sp++; reg ← mem[sp]
    // or sp−−

- Function call
  - call addr: push ip; ip ← addr
  - return: pop ip
swi $n$  ($n = 0, 1, \ldots, 4$)  // sw-interrupt; from cpu
   0: exception
   1–4: syscalls
   ...

hwi $n$  ($n = 5, \ldots, 9$)  // hw-interrupt; from external device
   5: timer
   6: io-adaptor
   ...

swi/hwi $n$
   push ip; push ps; ip ← mem[$n$]; ps ← intrpt-off, kernel-mode
   return: pop ps; pop ip

reset/power-up:  ip ← 0x100
IO Adaptor

- IO device similar to a disk
  - holds (multi-word) blocks at locations
  - io request by process: [r/w, location, addr-of-buffer]

- Data register dbr
  - holds a word to read or write

- Control register ctrl
  - op: read/write
  - loc: block’s location in device
  - addr: address of buffer in memory for block (if dma on)
  - intrpt: on/off; on → interrupt when operation done
  - dma: on/off
  - busy: true if operation ongoing // read-only
Outline

ToyOS: Hardware
ToyOS-1: no IO
ToyOS-2: Wait/Wakeup on PCB queue
Toy OS-3: IO: synchronous, no interrupt, no dma
Toy OS-4: IO: synchronous, interrupt, dma
Toy OS-5: IO: asynchronous, interrupt, dma
Memory at start

- Separate area for OS and for each user process
  
  - OS area:
    - data structures
    - functions
      - updateRunqPcb()
      - scheduler()
      - mem[0] → exceptionHndlr()  // swi 0: exception
    - kernel stack  // used by kernel code; no OS process for now

- Each user process
  - contiguous area: [low-address, high-address]
  - code, data, stack

ToyOS-1: no io
OS data structures

PCB (process control block): one per process
   // holds state of process when not running
   gpr initially arbitrary
   sp " " bottom of process stack
   ps " " intrpts on, mode user
   ip " " start addr of process code
   hi, lo " " high/low addr of process memory
   ioreq " " nil // io request, if any

Better: store ps and ip on stack instead of in PCB fields

PCBs circulate in two queues

runQ: // running process; at most 1 entry
readyQ: // ready processes; awaiting cpu

Initially all PCBs in readyQ
timerIntHndlr():  // here on timer interrupt

/* runQ holds pcb of interrupted process
   cpu.sp → stack top of runQ.pcb
   stack top has ps and ip values “at interrupt”
   cpu.ps: interrupts off, kernel mode
*/

updateRunQPcb()
move runQ.pcb to tail of readyQ
scheduler()

// return? from function? from interrupt?
updateRunqPcb(): // save the state of the interrupted process
/* Called from an interrupt (swi/hwi) handler
runQ holds pcb of last running process
cpu.sp → stack top of runQ pcb
stack top has ps and ip values “at interrupt”
cpu.ps: interrupts off, kernel mode
*/
runQpcb.gpr ← cpu.gpr
runQpcb.ip/ps ← ip/ps values on stack
runQpcb.sp ← cpu.sp // adjusted to “at interrupt”
cpu.sp ← kernel stack bottom // fresh start
push runQpcb.ip
return-from-function
scheduler(): // located at Reset address (0x100)

/* Wait for non-empty readyQ, dispatch process at head.
   Called from intrpt handler, runQ empty, intrpts disabled,
   cpu.sp → kernel stack */

while (readyQ empty) // busy wait with interrupts enabled
    cpu.ps ← interrupts on
    cpu.ps ← interrupts off
move pcb at readyQ.head to runQ
// dispatch runQ.pcb
cpu.gpr/sp/hi/lo ← runQ.pcb.gpr/sp/hi/lo
push runQ.pcb.ps/ip // using stack of process to be run
return-from-interrupt // pops ps and ip atomically
exceptionHndlr(): // here on execution; kill running process
remove runQ.pcb
delete pcb
scheduler()
// caller never comes here
ToyOS: Hardware
ToyOS-1: no IO
ToyOS-2: Wait/Wakeup on PCB queue
Toy OS-3: IO: synchronous, no interrupt, no dma
Toy OS-4: IO: synchronous, interrupt, dma
Toy OS-5: IO: asynchronous, interrupt, dma
Overview

- **User instructions:**
  - swi 1: syscall-wait(q)  
  - swi 2: syscall-wakeup(q)  

- **OS functions**
  - waitHndlr(q)  
  - wakeupHndlr(q)  

- **OS data structure**
  - mem[1] → waitHndlr(.)  
  - mem[2] → wakeupHndlr(.)  
  - PCB queue(s) on which to synchronize

ToyOS-2: wait/wakeup
Functions

- **waitHndlr(q):**
  
  ```
  updateRunqPcb
  move runQpcb to q
  scheduler()
  ```

- **wakeupHndlr(q):**
  
  ```
  // here on swi 1: syscall-wait(q)
  if (q not empty)
    move q.head.pcb to readyQ
  return-from-interrupt
  ```

ToyOS-2: wait/wakeup
Outline

ToyOS: Hardware
ToyOS-1: no IO
ToyOS-2: Wait/Wakeup on PCB queue
Toy OS-3: IO: synchronous, no interrupt, no dma
Toy OS-4: IO: synchronous, interrupt, dma
Toy OS-5: IO: asynchronous, interrupt, dma
Start from ToyOS-2

Add io capability to user process
- if io device is busy: process waits in an io queue
- if io device is not busy: process does io (accessing io adaptor); upon completion, wakes up a process (if any) waiting on io.

User instructions:
- swi 3: syscall-io(op, loc, addr)

OS data structure
- ioQ: PCB queue of processes with io requests, all waiting
- ioAvail: flag indicating whether io device is available

OS functions
- ioReqHndlr(op,loc,addr) // swi 3 handler
  // executed by user process in kernel mode with intrrpts on
ioReqHndlr(op, loc, addr): // here on swi 3, intrpt off, kernel

runQ.pcb.ioreq ← [op, loc, addr]
while (not ioAvail)
    swi 1 (ioQ): // syscall-wait(ioQ)
ioAvail ← false
set cpu.ps.intrpt on // share cpu
io-adaptor.ctrl ← [op, loc, addr, no intrpt, no dma]
for (j in 0 . . . blksize−1)
    while (io-adaptor.ctrl.busy) nop // busy wait, if op = r
    if (op = w) io-adaptor.db r ← mem[addr + j]
    else mem[addr + j] ← io-adaptor.db r
    while (io-adaptor.ctrl.busy) nop // busy wait, if op = w
ioAvail ← true
swi 2 (ioQ) // syscall-wakeup(ioQ); start next io, if any
return-from-interrupt
Outline

ToyOS: Hardware
ToyOS-1: no IO
ToyOS-2: Wait/Wake up on PCB queue
ToyOS-3: IO: synchronous, no interrupt, no dma
ToyOS-4: IO: synchronous, interrupt, dma
ToyOS-5: IO: asynchronous, interrupt, dma
IO Overview

- Start from ToyOS-2

- Add io capability to user process
  - process waits in an io queue, starting io if device is available
  - io intrpt handler wakes up process; starts new io (if any)

- User instructions:
  - swi 3: syscall-io(op, loc, addr)

- OS data structure
  - ioQ: PCB queue of processes with io requests
    - process at head (if any) is being served

- OS functions
  - ioReqHndlr(op,loc,addr) // swi 3 handler
  - ioIntHndlr() // hwi 6 handler
ioReqHndlr(op, loc, addr): // here on swi 3: syscall-io

runQpcb.ioreq ← [op, loc, addr]

if (ioQ empty) // io device not busy
    io-adaptor.ctrl ← [op, loc, addr, dma, intrpt]
swi 1 (ioQ) // syscall-wait(ioQ)

// return? from function? from interrupt?
IO Interrupt Handler

ioIntHndlr(): // here on hwi 6: io-adaptor interrupt

/* runQ holds pcb of interrupted process or is empty
   cpu.sp → stack top of runQ.pcb or of Kernel stack
   stack top has ps and ip values “at interrupt”
   cpu.ps: interrupts off, kernel mode
   ioQ is not empty: its head’s io request has just completed
*/

swi 1(ioQ) // syscall-wakeup(ioQ)
if (ioQ not empty) // start io for next waiting process
    io-adaptor.ctrl ← [ioQ.head.pcb.ioreq, dma, intrpt]
return-from-interrupt

/* This handler uses the interrupted process stack or kernel stack.
   Interrupt has nothing to do with interrupted process.
*/
Outline

ToyOS: Hardware
ToyOS-1: no IO
ToyOS-2: Wait/Wakeup on PCB queue
Toy OS-3: IO: synchronous, no interrupt, no dma
Toy OS-4: IO: synchronous, interrupt, dma
Toy OS-5: IO: asynchronous, interrupt, dma
Synchronous IO
- ioReqHndlr(x) returns only after x is served

Asynchronous IO
- ioReqHndlr(x) is non-blocking (returns “immediately”)
- ioReqQ: queue of io requests // not PCBs
- ioReqHndlr(x): adds x to ioReqQ; returns
- ioServer: kernel thread that serves requests from ioReqQ
- ioQ: now used only by ioServer
  - waits here for nonempty ioReqQ or io interrupt
Start from ToyOS-2

**User instructions:**
- swi 3: syscall-io(op, loc, addr)

**OS data structure**
- ioQ: PCB queue; holds at most 1 process (executing ioServer)
  - process at head (if any) waiting for io request/intrpt

**ioReqQ**

**ioServer’s PCB:** as usual except // actually TCB
- ps: intrpts-off, mode-kernel // so hi/lo irrelevant
- ip: points to ioServerFn()
Interrupt Handlers

- **ioReqHndlr(op, loc, addr):**  // here on swi 3: syscall-io
  add [op,loc,addr] to ioReqQ
  if (ioReqQ has 1 entry)
    swi 2(ioQ) // syscall-wakeup(ioQ); wake up ioServer
  return-from-interrupt

- **ioIntHndlr:**  // here on io interrupt
  /* ioQ has ioServer PCB only */
  syscall-wakeup(ioQ)
  return-from-interrupt
ioServerFn(): // executed by kernel thread “ioServer”
/* kernel mode, intrpts off, non terminating
while (true)
    if (ioReqQ empty) // note: ‘‘while’’ not needed
        swi 1(ioQ)    // syscall-wait(ioQ)
    // ioReqQ not empty
    if (io-adaptor.ctrl busy) // should not happen
        swi 1(ioQ)    // syscall-wait(ioQ)
    // ioReqQ not empty, io device not busy
    [op, loc, addr] ← ioReqQ.head    // need not be head
    io-adaptor.ctrl ← [op, loc, addr, intrpt, dma]
    swi 1(ioQ)    // syscall-wait(ioQ)
    remove ioReqQ.head
Suppose \( \text{ioReqQ} \) becomes full.

\( \text{ioReqQ} \) (and \( \text{ioQ} \) in synchronous IO) need not be FCFS. Can choose request to serve to optimize performance.

Disabling interrupts to protect OS resources is not desirable

- It blocks processes that do not need protected resources
- It works only in a single-processor system

More fine-grained mechanisms are needed