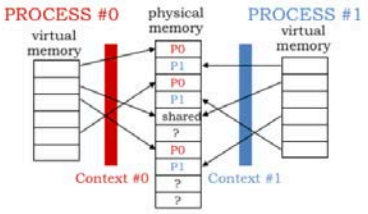
Computation Structures

Virtualizing the Processor Worksheet

Building a Virtual Machine (VM)

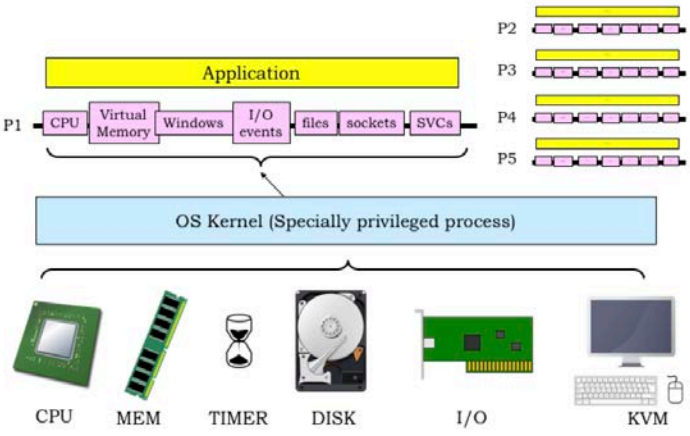


Goal: give each program its own "VIRTUAL MACHINE"; programs don't "know" about each other...

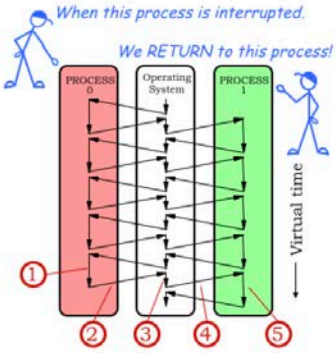
New abstraction: a process which has its own
- machine state: R0, ..., R30
- program (w/ shared code)
- context (virtual address space)
- virtual I/O devices
- PC, stack

"OS Kernel" is a special, privileged process running in its own context. It manages the execution of other processes and handles real I/O devices, emulating virtual I/O devices for each process.

One VM For Each Process



Processes: Multiplexing the CPU



1. Running in process #0
2. Stop execution of process #0 either because of explicit *yield* or some sort of timer *interrupt*; trap to handler code, saving current PC+4 in XP
3. First: save process #0 state (regs, context) Then: load process #1 state (regs, context)
4. "Return" to process #1: just like return from other trap handlers (ie., use address in XP) but we're returning from a *different* trap than happened in step 2!
5. Running in process #1

Interrupt Handler Coding

```
long TimeOfDay;
struct MState { int Regs[31];} UserMState;

/* Executed 60 times/sec */
Clock_Handler(){
   TimeOfDay = TimeOfDay+1;
   if (TimeOfDay % QUANTUM == 0) Scheduler();
}

Clock_h:
    ST(r0, UserMState)        // Save state of
    ST(r1, UserMState+4)      // interrupted
    ...                       //  app pgm...
    ST(r30, UserMState+30*4)
    LD(KStack, SP)            // Use KERNEL SP
    BR(Clock_Handler,lp)      // call handler
    LD(UserMState, r0)        // Restore saved
    LD(UserMState+4, r1)      //   state.
    ...
    LD(UserMState+30*4, r30)
    SUBC(XP, 4, XP)           // execute interrupted inst
    JMP(XP)                   // Return to app.
```

Handler (written in C)

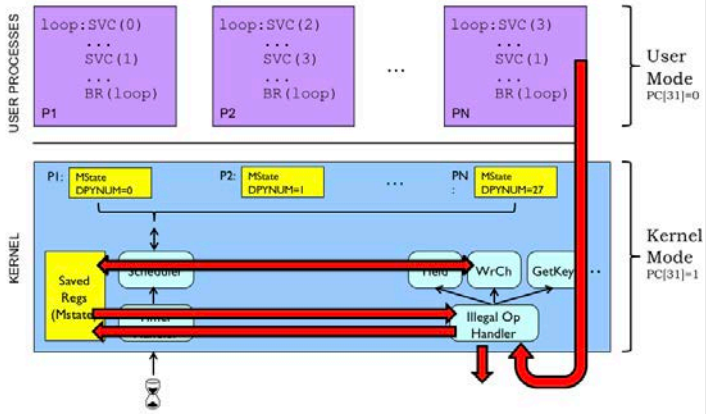
"Interrupt stub" (written in assy.)

Simple Timesharing Scheduler

```
struct MState { int Regs[31]; } UserMState;


struct PCB {       // Process Control Block
   struct MState State;       // Processor state
   struct Context PageMap;    // MMU state for proc
   int DPYNum;     // Console number (and other I/O state)
 } ProcTbl[N];                // one per process

int Cur;                      // index of "Active" process


Scheduler() {
   ProcTbl[Cur].State = UserMState; // Save Cur state
   Cur = (Cur+1)%N;                 // Incr mod N
   UserMState = ProcTbl[Cur].State; // Install state for next User
   LoadUserContext(ProcTbl[Cur].PageMap);  // Install context
}
```

Simple Timesharing Scheduler

OS Organization: Supervisor Calls



Problem 1.

In lecture we arrived at the following implementation for the ReadKey supervisor call, which waits until there is a character available in the keyboard buffer, then returns it to the user in R0. Three lines in the handler have been labeled [A], [B], and [C].

```
    ReadKey_h() {
      int kbdnum = ProcTbl[Cur].DPYNum;
      if (BufferEmpty(kbdnum)) {
[A]     User.Regs[XP] = User.Regs[XP]-4;
[B]     Scheduler();
      } else {
[C]     User.Regs[0] = ReadInputBuffer(kbdnum);
      }
    }
```

Below we'll consider the effect of removing each labeled line in turn. **Please choose one** of the following as the best characterization of the effect of removing the line:

1. Execution appears to halt as there is now a loop in Kernel mode.
2. Both the requesting process and other processes run as before.
3. The requesting process runs as before; other processes receive a smaller percentage of the CPU time.
4. The requesting process runs as before; other processes receive a larger percentage of the CPU time.
5. The requesting process receives an incorrect character.

(A) Line [A] is removed from the handler.

User resumes execution although no character was available.

Effect on execution? (circle one) 1 … 2 … 3 … 4 … (5)

(B) Line [B] is removed from the handler.

Keep re-trying Readkey although no character is available.

Effect on execution? (circle one) 1 … 2 … (3) … 4 … 5

(C) Line [C] is removed from the handler.

Forgot to put character into user's R0!

Effect on execution? (circle one) 1 … 2 … 3 … 4 … (5)

(D) A summer intern decides that if a process is waiting for a character it should be scheduled for execution half as often, so he adds a second call to `Scheduler()`, i.e., he duplicates line [B]. Briefly describe the actual effect of this change. To be concrete assume there are N processes and that it's process 0 that executes the ReadKey SVC.

Brief description

Process #1 is never scheduled while Process #0 is waiting for a character.

Problem 2.

A Beta running the OS from lab 8 is running two processes:

```
// Process 0:              // Process 1:
P0:   GetKey()             P1:   ADDC(R0, 1, R0)
      WrCh()
      BR(P0)                     BR(P1)
```

XP always points here when execution resumes.

You type a few characters and see them echoed.

(A) What can you say about the value in the XP register on return from the **`GetKey()`** SVC?

It may have different values on consecutive returns: TRUE **FALSE**

Since the last statement is true, all the other statements are also true.

It always has a high-order 0 bit: **TRUE** FALSE

It is always a multiple of 4: **TRUE** FALSE

It always has the value P0+4: **TRUE** FALSE

You notice that Process 1 increments R0, whose value increases at some fairly constant rate **R** increments/second. You experiment with several changes below, not typing but monitoring the rate at which the count in Process 1's R0 increases. For each of the experiments below, you are asked to estimate its effect on the R0 counting rate, relative to **R**; choose among

+LOTS: the rate increases considerably, e.g., twice **R**.
SAME: the rate is about the same as **R**
-LOTS: the rate is much lower, e.g., **R/2** or lower.

(B) As an experiment, you eliminate Process 0 (so that only Process 1 is running). How does the rate of increase of Process 1's R0 change? Assume the scheduler time slice for each process is long compared to one iteration of either loop.

P0 usually calls Scheduler (mostly ~~[illegible]~~ there's no character to return). So running R0 takes very little time.

Circle one: +LOTS **SAME** -LOTS

You restore both processes and *eliminate the Scheduler() call* invoked by the GetKey() SVC when no character is ready to be returned.

(C) How does this effect the rate at which R0 increases, relative to the original counting rate **R**?

P0 keeps looping, calling GetKey. Only yields to P1 at end of timesharing slice.

Circle one: +LOTS SAME **-LOTS**

(D) Again running both Process 0 and Process 1, you now replace the single Scheduler() call invoked by GetKey() by **two consecutive** Scheduler() calls. How does the rate at which Process 1 counts change, relative to the original rate **R**?

See 1(C)!

Circle one: +LOTS SAME **-LOTS**

Problem 3.

Real Virtuality, Inc. markets three different computers, each with its own operating system. The systems are:

Model A: A timeshared Beta system whose OS kernel is uninterruptable.
Model B: A timeshared Beta system which enables device interrupts during handling of SVC traps.
Model C: A single-process (not timeshared) system which runs dedicated application code.

Each system runs an operating system that supports concurrent I/O on several devices, including an operator's console with a keyboard. Les N. Dowd, RVI's newly-hired OS expert, is in a jam: he has dropped the shoebox containing the master copies of OS source for all three systems. Unfortunately, three disks containing handlers for the ReadKey SVC trap, which reads and returns the ASCII code for the next key struck on the keyboard, have gotten confused. Of course, they are unlabeled, and Les isn't sure which handler goes into the OS for which machine. The handler sources are

```
ReadCh_h() {                                /* VERSION R1 */
  if (BufferEmpty(0))                       /* Has a key been typed? */
    User->Regs[XP] = User->Regs[XP]-4;      /* Nope, wait.       */
  else
    User->Regs[0] = ReadInputBuffer(0); /* Yup, return it. */
}

ReadCh_h() {                                /* VERSION R2 */
  int kbdnum=ProcTbl[Cur].KbdNum;
  while (BufferEmpty(kbdnum)) ;             /* Wait for a key to be hit*/
  User->Regs[0] = ReadInputBuffer(kbdnum);  /*...then return it. */
}

ReadCh_h() {                                /* VERSION R3 */
  int kbdnum=ProcTbl[Cur].KbdNum;
  if (BufferEmpty(kbdnum)) {                /* Has a key been typed? */
    User->Regs[XP] = User->Regs[XP]-4;      /* Nope, wait. */
    Scheduler();
  } else
    User->Regs[0] = ReadInputBuffer(kbdnum);   /* Yup, return it. */
}
```

(A) Show that you're cleverer than Les by figuring out which handler goes with each OS, i.e., for each operating system (A, B and C) indicate the proper handler (R1, R2 or R3).

Model A goes with handler (circle one): R1 ... R2 ... R3

Model B goes with handler (circle one): R1 ... R2 ... R3

Model C goes with handler (circle one): R1 ... R2 ... R3

But Les isn't that smart. In order to figure out which handler code goes with each OS version, Les makes copies of each disk and distributes them as "updates" to beta-test teams for each OS. Les figures that if each handler version is tried by some beta tester in each OS, the comments of the testers will allow him to determine the proper OS for each handler.

Les sends out the alleged source code updates, routing each handler source to testers for each OS. In response, he gets a barrage of complaints from many of the testers. Of course, he's forgotten which disk he sent to each tester. He asks your help to figure out which combination of system and hander causes each of the complaints.

For each complaint below, explain which handler and which OS the complainer is trying to use.

(B) Complaint: "I get compile-time errors; Scheduler and ProcTbl are undefined!"

User has handler R3 **on system** C

(C) Complaint: "Hey, now the system always reads everybody's input from keyboard 0. Besides that, it seems to waste a lot more CPU cycles than it used to."

User has handler R1 **on system** A

(D) (Complaint: "Neat, the new system seems to work fine. It even seems to waste less CPU time than it used to!"

User has handler R3 **on system** B

Problem 4.

The Yield() SVC can be used in user-mode programs on a time-sharing system to give up the remainder of their current time slice. The kernel implementation of the Yield simply calls the kernel Scheduler() routine to choose another process to execute. When the yielding process is next scheduled, user-mode execution resumes with the instruction following the Yield() SVC.

Complete the code for the handler for a new SVC, YieldN(), which expects a numeric value, N, in the user's R0 and behaves as if the user program had contained N consecutive Yield() SVCs. When execution resumes following the completion YieldN(), R0 should contain 0.

```
YieldN_h() {
    if (User.Regs[0] > 0) {
        User.Regs[0] -= 1;    // decrement count
        User.Regs[XP] -= 4;   // arrange to re-execute SVC
        Scheduler();          // yield to next process
    } else {

        User.Regs[0] = 0;     // [optional] final value for R0

    }
}
```

Problem 5.

BetaSoft, Inc, the leading provider of Beta OS software, sells an operating system for the Beta similar to that described in lecture. It uses a simple round-robin scheduler, and has no virtual memory -- all processes share a single address space with the kernel, much like the OS of lab 8. The OS timeshares the Beta CPU among N processes using a simple, familiar scheduler shown below.

```
struct MState { int Regs[31]; } User;

struct PCB {              // Process Descriptor Block
  struct MState State;    // Saved process state
  ...;                    // Possible other stuff
} ProcTbl[N];             // One PCB per process

int Cur = 0;

Scheduler() {
  ProcTbl[Cur].State = User;
  Cur = (Cur+1) % N;
  User = ProcTbl[Cur].State;
}
```

Several of Betasoft's customers use the Beta for long, compute-bound applications, and have asked for a tool to help them find where their programs are spending most of their time. To accommodate these requests, BetaSoft has implemented a supervisor call, **SamplePC**, which allows a diagnostic program running in one process to sample the values in the PC of another. Betasoft proposes to write such a program, called a profiler, that takes many samples of PC values from a running program and produces a revealing histogram.

The SamplePC SVC takes a process number p in R0, and returns in R1 the value currently in the program counter of process p. The C portion of the SVC handler is given below:

```
SamplePC_h() {
  int p = User.Regs[0];
  int pc = ProcTbl[p].State.Regs[XP];
  ??? = pc;                          // incomplete code!
}
```

(A) Give the missing code fragment shown above as **???**.

User.Regs[1]

(write missing fragment of C code)

BetaSoft writes a simple profiler using the above SVC and uses it to measure a compute-bound process consisting of a single 10000-instruction loop. Noticing a surprisingly large number of repeated values in the sampled PC data, they cleverly deduce that their profiler is making many **SamplePC** calls during each time quanta for which the profiling process is scheduled.

returning redundant samples from the process being measured.

(B) Suggest a simple change to the **SamplePC_h** code that eliminates the observed problem. Be specific.

add Scheduler() call

(Describe simple modification to SamplePC_h code)

BetaSoft ignores your solution (keeping the original **SamplePC_h** code), arguing that they'll just collect enough samples that the redundant values won't affect the histogram significantly. They produce a working profiler program that takes many samples of another process's PC and produces a histogram showing code "hot spots". Although the profiler proves useful on compute-intensive application code, BetaSoft tries running it on a simple echo loop running in a process:

```
| echo loop, as a test for profiler tool:
.=0x100              | Test program starts at hex 100
loop:  GetKey()      | SVC: read char into R0
       WrCh()        | SVC: type char from R0
       BR(loop)      | ... and keep doing it!
```

(C) When run on the above process, what does the profiler report as the most common value of the PC? Answer "None" if you can't tell.

Often-reported PC value, or "None": 0x 100

The final BetaSoft profiler program itself is mostly a big loop, consisting of a single SamplePC SVC instruction located at **0x1000**, plus lots of compute-intensive additional code to appropriately format the collected data and write it into a file. Out of curiosity, BetaSoft engineers run the profiler in process 0, and ask it to generate a histogram of sampled PC values for process 0 itself.

(D) Which of the following best summaries their findings?

(1) All of the sampled PC values point to kernel OS code.
(2) The sampled PC is always **0x1004**.
(3) The SamplePC call never returns.
(4) None of the above.

Give number of best answer: 4

SamplePC returns XP saved in the process table, which is only updated at the call to Scheduler when switching to next process.

→ It does not read User.MState.

6.004 Computation Structures
Spring 2017