

Operating System Kernel

More Virtual Stuff

Why an OS?

What we've got:

- A Single Sequence Machine, capable of doing ONE thing at a time – one instruction, one I/O operation, one program.
- A universe of gadgets – e.g. I/O devices – that do similar things slightly differently.

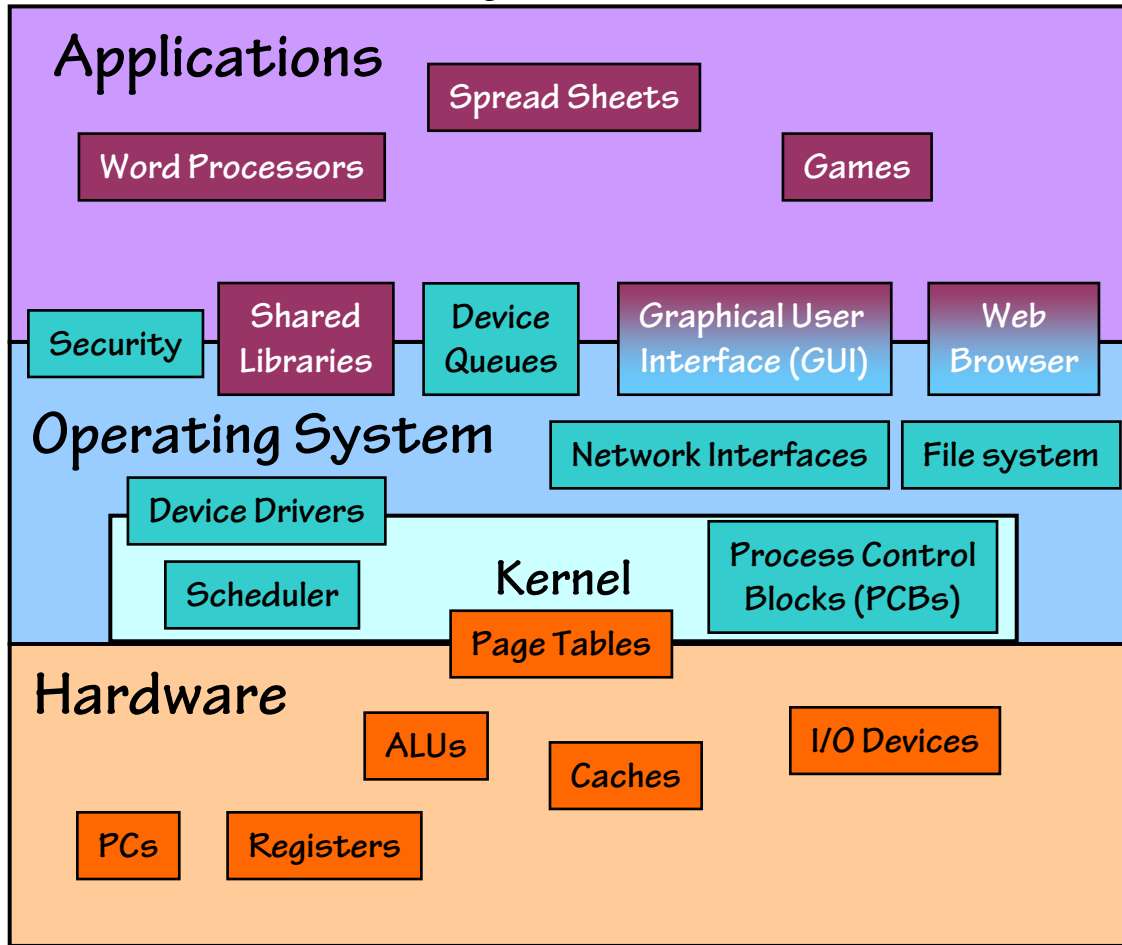
What we'd like:

- To listen to MP3s while reading email.
- To access disk, network, and screen “simultaneously”.
- To write a single program that does I/O with anybody's disk.

Plausible approaches:

- An infinite supply of identical computers with uniform, high-level peripherals for every conceivable purpose... or
- An illusion: Make one real computer look like many “virtual” ones.

Operating Systems



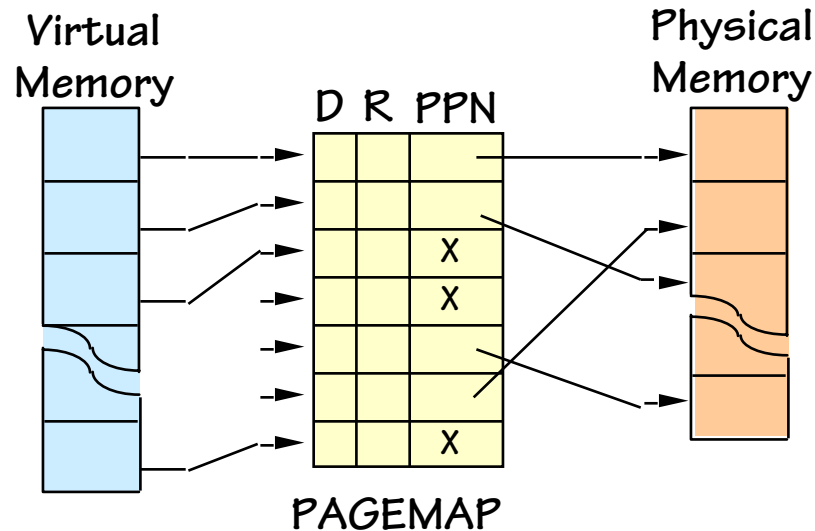
An OS is the Glue that holds a computer together.

- Mediates between competing requests
- Resolves names/bindings
- Maintains order/fairness

KERNEL - a RESIDENT portion of the O/S that handles the most common and fundamental service requests.

vir.tu.al \v*rch-(*-)w*l, 'v*r-ch*\ \.v*r-ch*-'wal-*t-e-\ \v*rch-(*-)w*-le-, 'v*rch-(*-)le-\ aj [ME, possessed of certain physical virtues, fr. ML virtualis, fr. L virtus strength, virtue : being in essence or effect but not in fact - vir.tu.al.i.ty n

Review: Virtual Memory

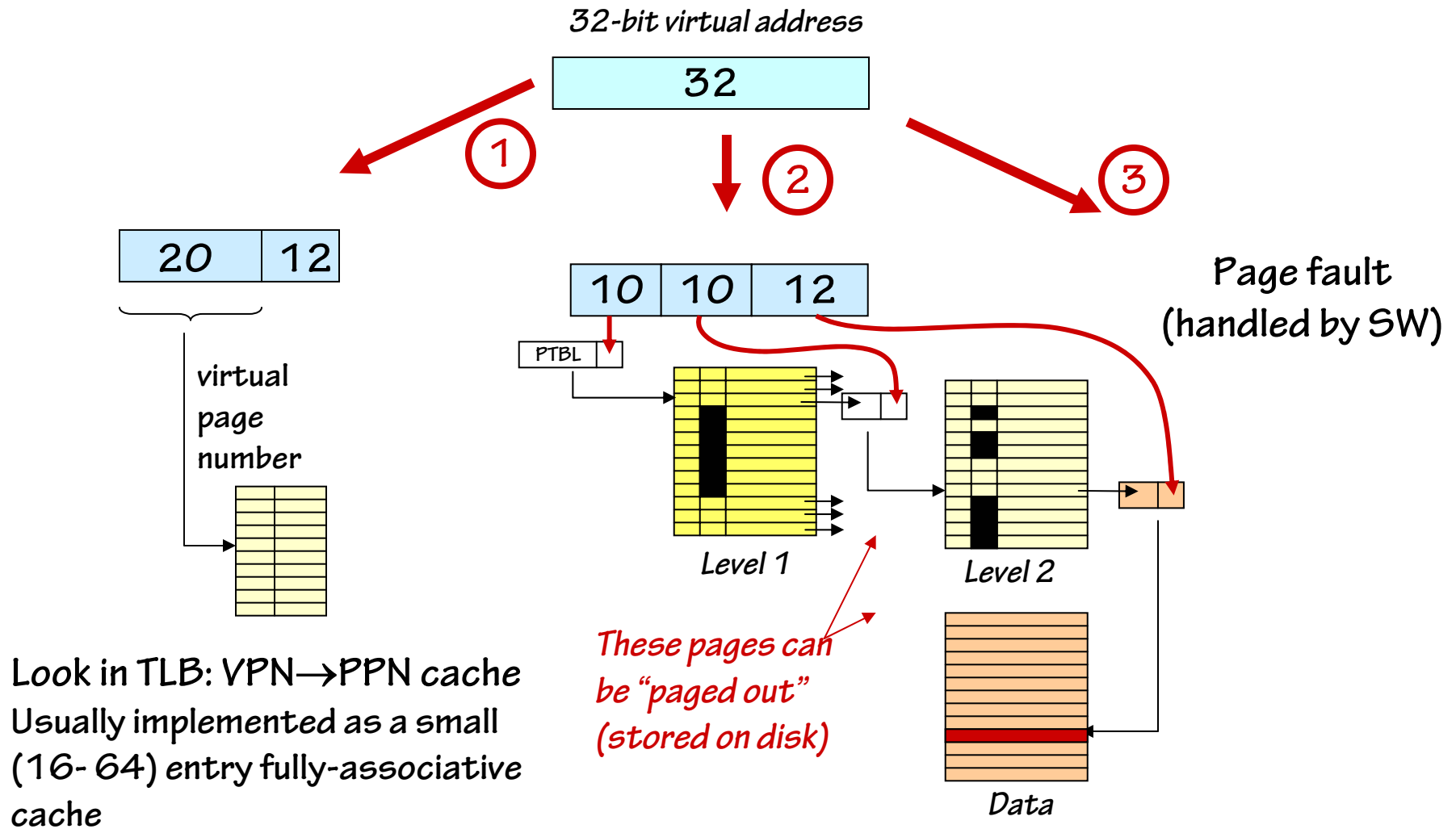


Goal: create illusion of large virtual address space

- divide address into (VPN,offset), **map** to (PPN,offset) **or page fault**
- **use high address** bits to select page: keep related data on same page
- use cache (**TLB**) to speed up mapping mechanism—works well
- **long disk latencies**: keep working set in physical memory, use write-back

MMU Address Translation

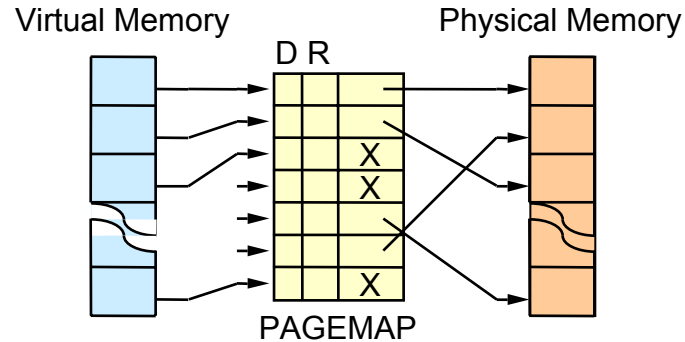
Typical Multi-level approach



Contexts

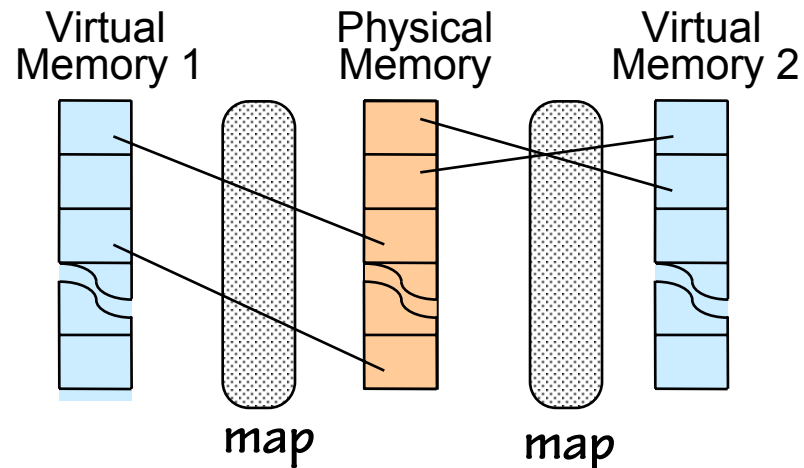
A context is an entire set of mappings from VIRTUAL to PHYSICAL page numbers as specified by the contents of the page map:

We might like to support multiple VIRTUAL to PHYSICAL Mappings and, thus, multiple Contexts.

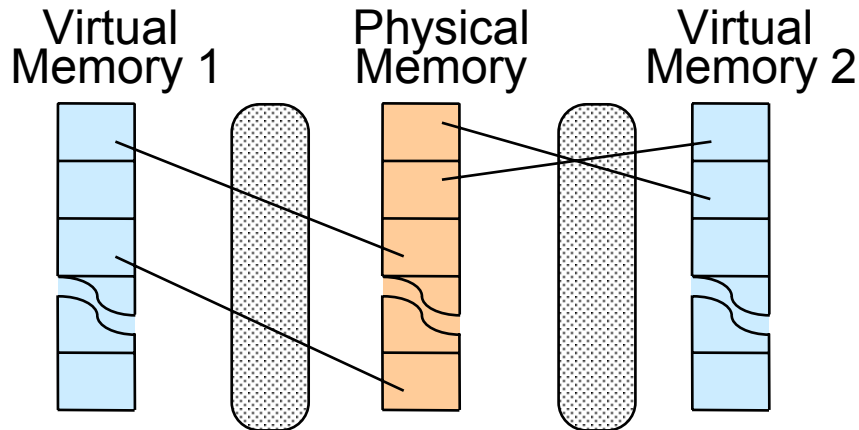


THE BIG IDEA: Several programs, each with their own context, may be simultaneously loaded into main memory!

“Context switch”:
reload the page map!



Power of Contexts: Sharing a CPU



Every application can be written as if it has access to all of memory, without considering where other applications reside.

More than Virtual Memory:
A VIRTUAL MACHINE

1. TIMESHARING among several programs --

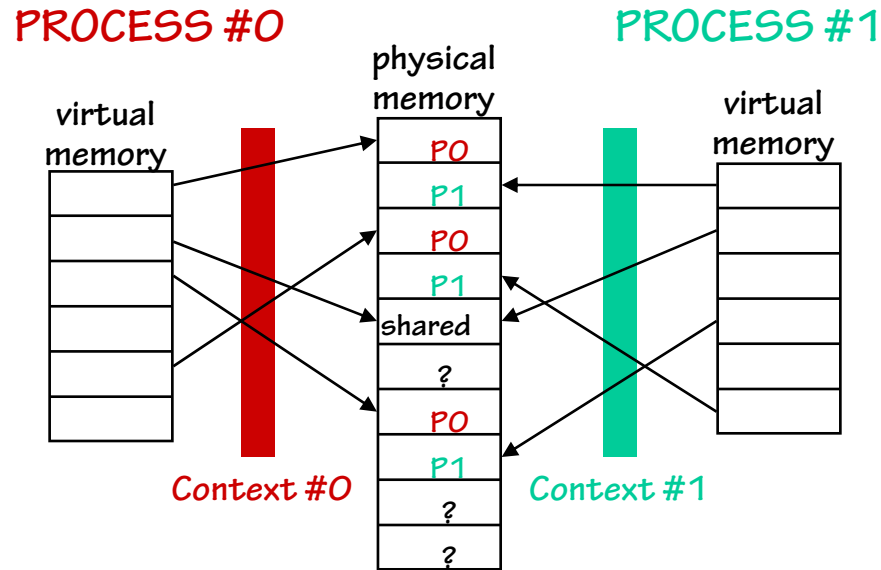
- Separate context for each program
- OS loads appropriate context into pagemap when switching among pgms

2. Separate context for OS “Kernel” (eg, interrupt handlers)...

- “Kernel” vs “User” contexts
- Switch to Kernel context on interrupt;
- Switch back on interrupt return.

TYPICAL HARDWARE SUPPORT: 2 HW pagemaps

Building a Virtual Machine



Goal: give each program its own “VIRTUAL MACHINE”; programs don’t “know” about each other...

Abstraction: create a **process** which has its own

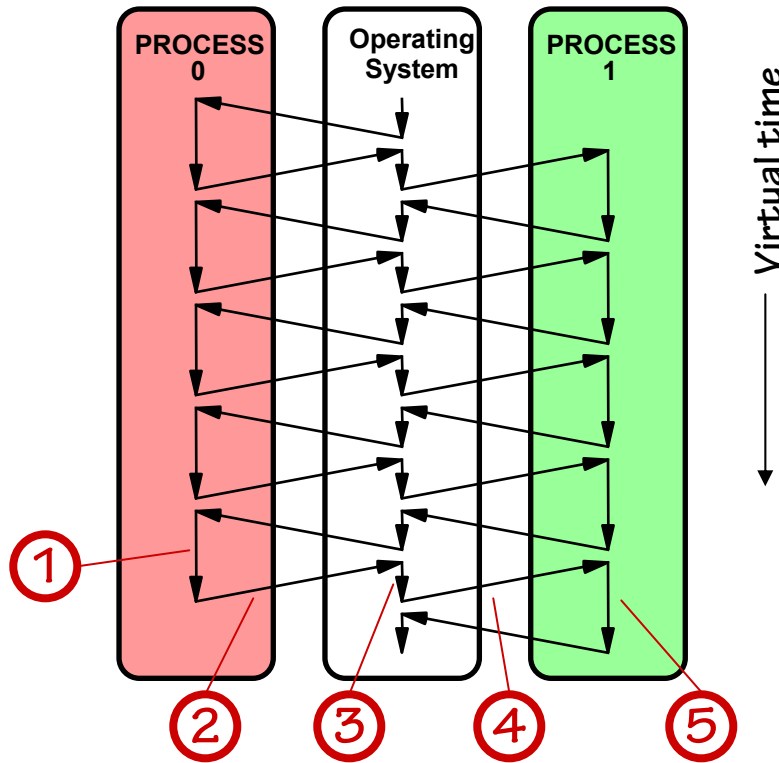
- machine state: R0, ..., R30
- context (virtual address space)
- stack
- program (w/ shared code)
- virtual I/O devices (console...)

Processes:

Multiplexing the CPU

When this process is interrupted.

We RETURN to this process!



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 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

Key Technology: Interrupts.

Interrupt Handling:

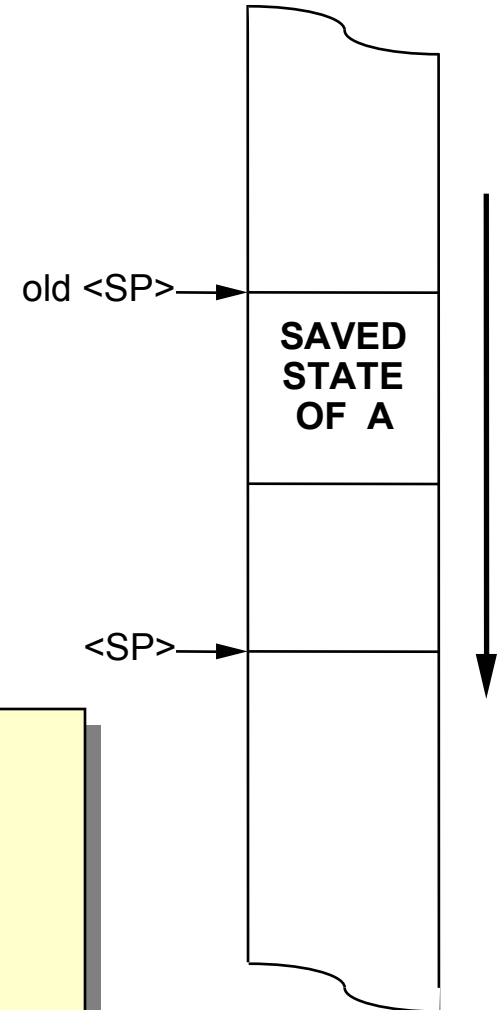
A primitive, constrained form of multiprocessing

BASIC SEQUENCE:

- Program A is running when some EVENT happens.
- PROCESSOR STATE saved on stack (like CALL)
- The HANDLER program to be run is selected.
- HANDLER state (PC, etc) installed as new processor state.
- HANDLER runs to completion
- State of interrupted program A popped from stack and re-installed, JMP returns control to A
- A continues, unaware of interruption.

CHARACTERISTICS:

- *TRANSPARENT* to interrupted program!
- Handler runs to completion before returning
- Obeys stack discipline: handler can "borrow" stack from interrupted program (and return it unchanged) or use a special handler stack.



External (Asynchronous) Interrupts

Example:

System maintains current time of day (TOD) count at a *well-known* memory location that can be accessed by programs. But...this value must be updated periodically in response to clock EVENTS, i.e. signal triggered by 60 Hz clock hardware.

Program A (Application)

- Executes instructions of the user program.
- Doesn't want to know about clock hardware, interrupts, etc!!
- Can incorporate TOD into results by examining well-known memory location.

Clock Handler

- GUTS: Sequence of instructions that increments TOD. Written in C.
- Entry/Exit sequences save & restore interrupted state, call the C handler. Written as assembler “stubs”.

Interrupt Handler Coding

```
long TimeOfDay;
struct Mstate { int R0, ..., R30; } User;

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

Handler
(written in C)

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

“Interrupt stub”
(written in assy.)

Simple Timesharing Scheduler

```
struct Mstate {                               /* Structure to hold */
    int Regs[31];                             /* processor state   */
} User;

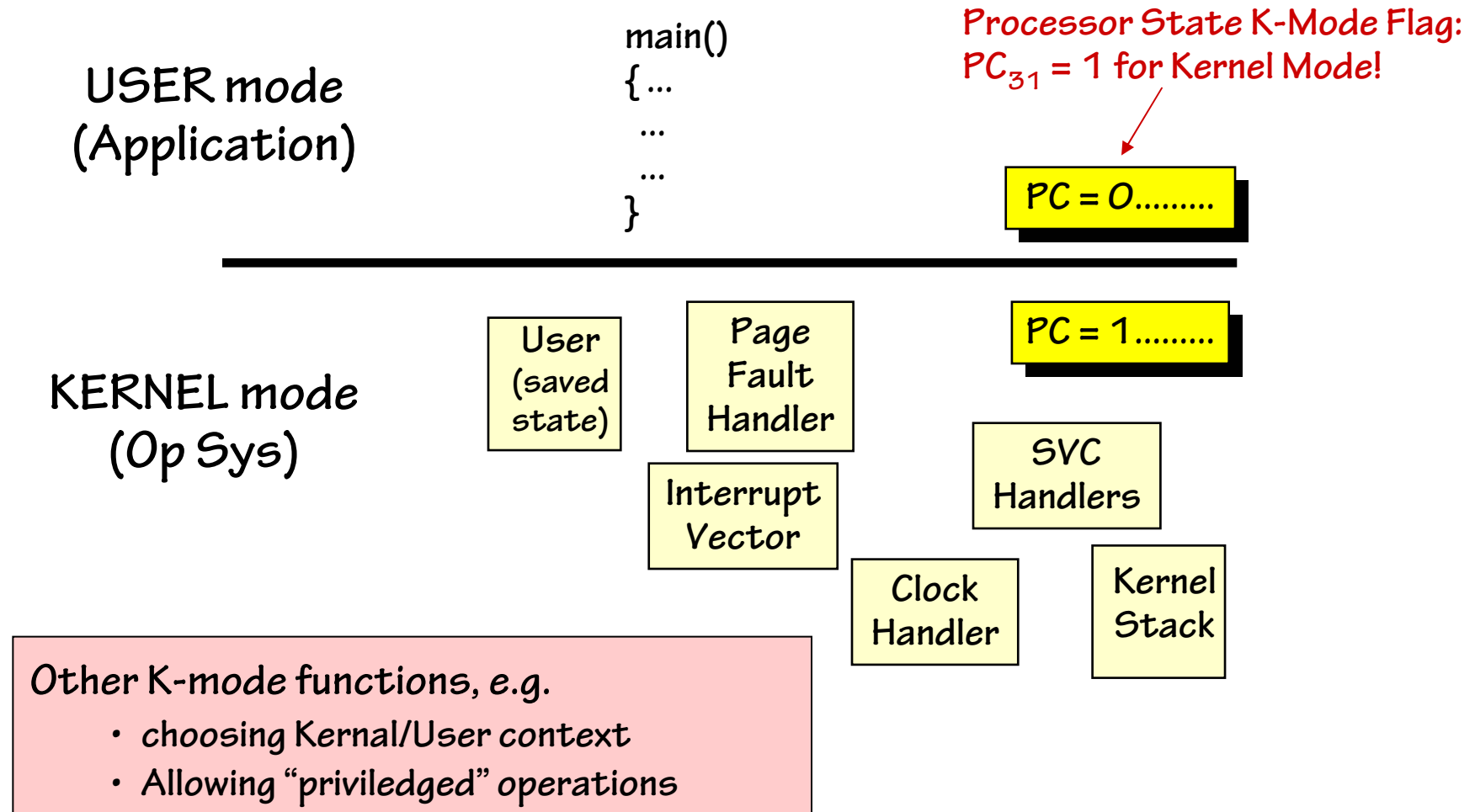
        (PCB = Process Control Block)
struct PCB {
    struct MState State;                      /* Processor state   */
    Context PageMap;                          /* VM Map for proc   */
    int DPYNum;                               /* Console number    */
} ProcTbl[N];                                /* one per process   */

int Cur;                                     /* "Active" process */

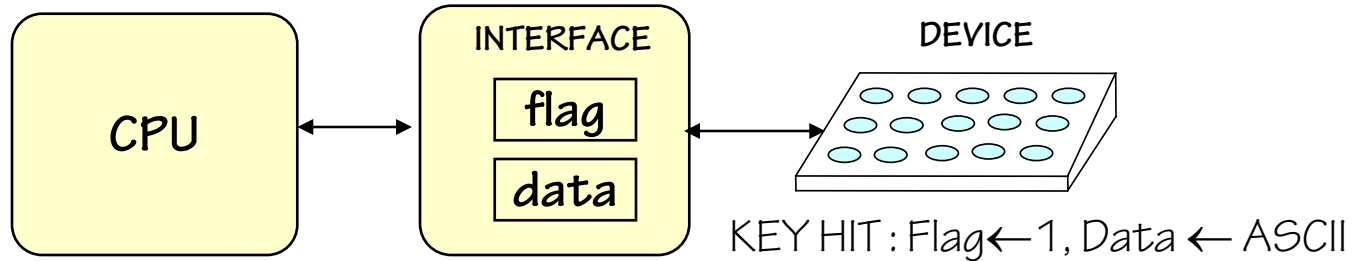
Scheduler() {
    ProcTbl[Cur].State = User;               /* Save Cur state   */
    Cur = (Cur+1)%N;                         /* Incr mod N       */
    User = ProcTbl[Cur].State; /* Install state for next User */
    LoadUserContext(ProcTbl[Cur].Context); /* Install context  */
}
```

Avoiding Re-entrance

Handlers which are interruptable are called *RE-ENTRANT*, and pose special problems... **Beta, like many systems, disallows reentrant interrupts!**
Mechanism: Uninterruptable “Kernel Mode” for OS:



Polled I/O



Application code deals directly with I/O (eg, by busy-waiting):

```
loop: LD(flag, r0)
      BEQ(R0, loop)
      ...      | process keystroke
```

PROBLEMS:

- Uses up (physical) CPU while busy-waiting
 - (FIX: Multiprocessing, codestripping, etc)
- Poor system modularity: running pgm MUST know about ALL devices.
- Uses up CPU cycles even when device is idle!

Interrupt-Driven I/O

OPERATION: NO attention to Keyboard during normal operation

- on key strike: hardware asserts IRQ to request interrupt
- USER program interrupted, PC+4 of interrupted inst. saved in XP
- state of USER program saved on KERNEL stack;
- KeyboardHandler invoked, runs to completion;
- state of USER program restored; program resumes.

TRANSPARENT to USER program.

Keyboard Interrupt Handler (in O.S. KERNEL):

Assume each
keyboard has
an associated
buffer

```
struct Device {
    char Flag, Data;
} Keyboard;

KeyboardHandler(struct Mstate *s) {
    Buffer[inptr] = Keyboard.Data;
    inptr = (inptr + 1) % 100;
}
```

ReadKey SVC: Attempt #1

A *supervisor call* (SVC) is an instruction that transfers control to the kernel so it can satisfy some user request. Kernel returns to user program when request is complete.

First draft of a ReadKey SVC handler: returns next keystroke to user

```
ReadKEY_h ()
{
    int kbdnum = ProcTbl[Cur].DPYNum;
    while (BufferEmpty(kbdnum)) {
        /* busy wait loop */
    }
    User.Reg[0] = ReadInputBuffer(kbdnum);
}
```

Problem: Can't interrupt code running in the supervisor mode...
so the buffer never gets filled.

ReadKey SVC: Attempt #2

A keyboard SVC handler (slightly modified, eg to support a Virtual Keyboard):

```
ReadKEY_h ()
{
    int kbdnum = ProcTbl[Cur].DPYNum;
    if (BufferEmpty(kbdnum)) {
        User.Reg[XP] = User.Reg[XP]-4;
    } else
        User.Reg[0] = ReadInputBuffer(kbdnum);
}
```

That's a
funny way
to write
a loop



Problem: The process just wastes its time-slice waiting for some one to hit a key...

ReadKey SVC: Attempt #3

BETTER: On I/O wait, YIELD remainder of quantum:

```
ReadKEY_h()
{
    int kbdnum = ProcTbl[Cur].DPYNum;
    if (BufferEmpty(kbdnum)) {
        User.Reg[XP] = User.Reg[XP]-4;
        Scheduler( );
    } else
        User.Reg[0] = ReadInputBuffer(kbdnum);
}
```

RESULT: Better CPU utilization!!

Does timesharing causes CPU use to be less efficient?

- COST: Scheduling, context-switching overhead; but
- GAIN: Productive use of idle time of one process by running another.

Sophisticated Scheduling

To improve efficiency further, we can avoid scheduling processes in prolonged I/O wait:

- Processes can be in **ACTIVE** or **WAITING** (“sleeping”) states;
- Scheduler cycles among **ACTIVE PROCESSES** only;
- Active process moves to **WAITING** status when it tries to read a character and buffer is empty;
- Waiting processes each contain a code (eg, in PCB) designating what they are waiting for (eg, keyboard N);
- Device interrupts (eg, on keyboard N) move any processes waiting on that device to **ACTIVE** state.

UNIX kernel utilities:

- `sleep(reason)` - Puts CurProc to sleep. “Reason” is an arbitrary binary value giving a condition for reactivation.
- `wakeup(reason)` - Makes active any process in `sleep(reason)`.

ReadKey SVC: Attempt #4

```
ReadKEY_h() {  
  ...  
  if (BufferEmpty(kbdnum)) {  
    User.Regis[XP] = User.Regis[XP] - 4;  
    sleep(kbdnum);  
  }  
  ...  
}
```

SVC call from application

```
sleep(status s) {  
  ProcTbl[Cur].status = s;  
  Scheduler()  
}
```

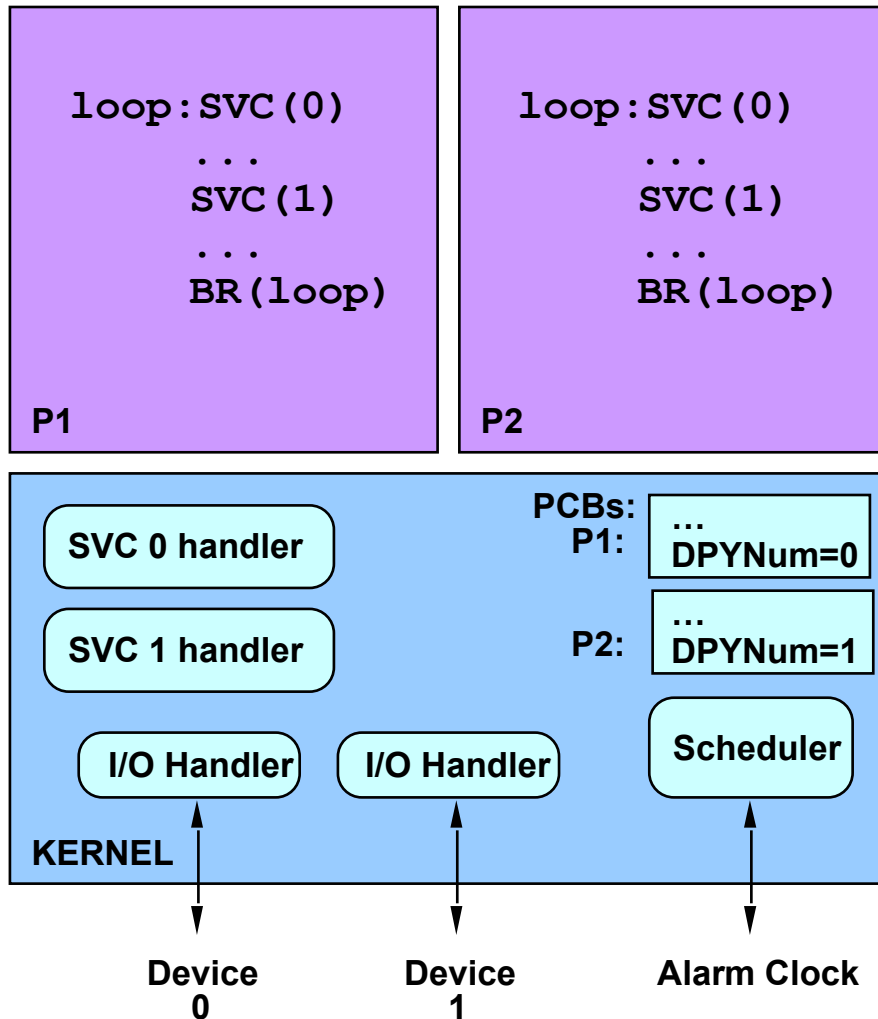
```
Scheduler() {  
  ...  
  while (ProcTbl[i].status != 0) {  
    i = (i+1)%N;  
  }  
  ...  
}
```

```
wakeup(status s) {  
  for (i = 0; i < N; i += 1) {  
    if (ProcTbl[i].status == s)  
      PCB[i].status = 0;  
  }  
}
```

```
KEYhit_h() {  
  ...  
  WriteBuffer(kbdnum, key)  
  wakeup(kbdnum);  
  ...  
}
```

INTERRUPT from Keyboard n

OS organization



“Applications” are quasi-parallel
“PROCESSES”

on
“VIRTUAL MACHINES”,
each with:

- CONTEXT
- (virtual address space)
- Virtual I/O devices

O.S. KERNEL has:

- Interrupt handlers
- SVC (trap) handlers
- Scheduler
- PCB structures containing the state of inactive processes

Summary

Virtual Stuff –

- Memory
- I/O
- Instructions
- CPU

Implementation Technology: Interrupts

- *Can share stack – obeys stack discipline!*
- *Transparent (well, except for XP...)*
- *Handler coding*

Timesharing

- *Separate stack / process – they don't obey stack discipline!*
- *Coding trick: Interrupt process A, return to process B – changing stack (or context) in the scheduler!*