# EECE7376: Operating Systems Interface and Implementation

xv6 and x86 Overview


# xv6 Overview

- **xv6** is a lightweight operating system, which is a reimplementation of the 6th Edition Unix, written in ANSI C with few parts written in assembly code.
- xv6 targets the x86 and RISC-V computer architectures.
- It was created for educational purpose in an MIT's OS Engineering course. The time to **compile** it is very low as it lacks some functionality that you would expect from a modern operating system.
- You can boot xv6 on real hardware, but typically we run it using the **QEMU** x86 emulator.


# QEMU Overview

- QEMU (short for **Quick Emulator**) is a virtual machine (VM) that we can use to emulate the x86 computer architecture.
  - x86 is an Intel instruction set architecture initially developed for the 8086 microprocessor and its 8088 variant.
- We will use this VM to compile and run the xv6 operating system.
- This allows us to check the implementation of the xv6 system calls and user commands programs with the option to test new ideas by modifying these programs.

# Installing xv6 in the COE Linux gateway

- QEMU is already installed in the CoE **nu** machine.
- Procedures on How to log into the COE Linux gateway: https://wiki.coe.neu.edu/xwiki/bin/view/Main/Linux_Machine_Help/
- If you do not have access to a COE Linux Account, check: https://coe.northeastern.edu/computer/general-services/request-a-coe-ece-computer-account/
- After logging to the COE Linux gateway and to install xv6 in your account, run the following commands:

1. `ssh -p27 nu`

   *if prompt for a password, enter your CoE password*

2. `git clone git://github.com/mit-pdos/xv6-public.git`

- If step 2 gets a Connection timed out error, try:

```
git clone http://github.com/mit-pdos/xv6-public.git
```

# Installing QEMU & xv6 Locally

- Use the following steps to install QEMU and xv6 in your personal computer Linux installation.

```
1. sudo apt-get update
2. sudo apt-get install build-essential
3. sudo apt-get install gcc-multilib
4. sudo apt install qemu qemu-utils qemu-kvm libvirt-clients bridge-utils
5. sudo apt-get install git-core
6. sudo git clone git://github.com/mit-pdos/xv6-public.git
7. cd xv6-public
8. sudo make
9. sudo apt install qemu-system-x86
```

- If step 6 gets a Connection timed out error, try:

```
sudo git clone http://github.com/mit-pdos/xv6-public.git
```

# Running xv6

- From a **Linux shell** where both QEMU VM and xv6 are installed (for the CoE server you need first to run `ssh -p27 nu`) use the following commands to start booting the xv6 OS on the VM:

```
$ cd xv6-public
$ sudo make qemu-nox
```

  - ***Note:*** `sudo` (for super user do) is not required if you are working from the CoE Linux server. `nox` stands for no graphics
- Now you are running the VM with the xv6 OS. We will refer to this shell as the **VM/xv6 shell**.
- To turn off the VM, press `Ctrl+A` followed by **`C`**. This will display a QEMU prompt. Here, type command "`q`", and press **Enter**
- To rebuild xv6, go to the `xv6-public` directory in the Linux shell and run the following command:

```
$ sudo make clean
```


# Windows Subsystem for Linux

- You can install Linux on top of Windows by installing Windows Subsystem for Linux (WSL) and then the Ubuntu Linux distribution:
  - https://docs.microsoft.com/en-us/windows/wsl/install
  - https://ubuntu.com/tutorials/ubuntu-on-windows#1-overview


# Modifying an xv6 User Command

- One of the xv6 user commands is **`ls`**.
- When you run `ls` from the **`VM/xv6`** shell, the contents of the root directory will be displayed. Each entry contains the following four fields:
  1. The item name
  2. Its type (1 = directory, 2 = file, 3 = device)
  3. The inode number, which refers to a data structure that contains the file metadata (will be discussed in the Persistence slides).
  4. The item size.
- From the Linux shell (not from the VM/xv6 shell) you can, for example, change how items are displayed by modifying `ls.c` (located in the **`xv6-public`** folder).
- Next time you start the VM/xv6 shell (using `sudo make qemu-nox`) and run **`ls`**, the new changes will take place.


# Editing WSL Ubuntu Files

- You will need **`sudo`** to edit files in your local WSL Ubuntu setup.
- To edit WSL Ubuntu files from Windows (e.g., using Notepad++) do the following:
  1. From Ubuntu prompt, grant read/write permissions on the files you need to edit to other (users over a network) and group: `sudo chmod og=rw <filename>`
  2. Open the files in your Windows editor from the following folder: `\\wsl.localhost\Ubuntu\home\<user>`
- *Note:* the reason you need to change the files permission is that the above folder is considered a network folder for WSL.


# Adding New xv6 User Command (1 of 2)

- To add a new user command to xv6 and make it available at the shell prompt, you need to add the C code of the new command and add an entry for it in the xv6 `Makefile`.
  - User commands can run from anywhere not like regular programs where you must run them from their folder.
- For example, in the `xv6-public` folder from the Linux shell, create a new file named `hello.c`, with the following content:

```
#include "types.h"
#include "user.h"
int main(int argc, char **argv)        {
 printf(1, "Hello world\n");
 exit(); //don't use return                }
```

- *Notes:*
  - The standard *C* library is not available in xv6. Instead, some utility sources and headers provide their own versions of common functions, such as printf, or exit)
  - In xv6, all user space applications must use `exit()` and not simply **return** from main.

# Adding New xv6 User Command (2 of 2)

- To add **hello** to the list of current user commands, edit file **Makefile** and add an entry named **_hello** to the end of the currently assigned values to the **UPROGS** variable:

```
UPROGS=\
        …
        _hello\
```

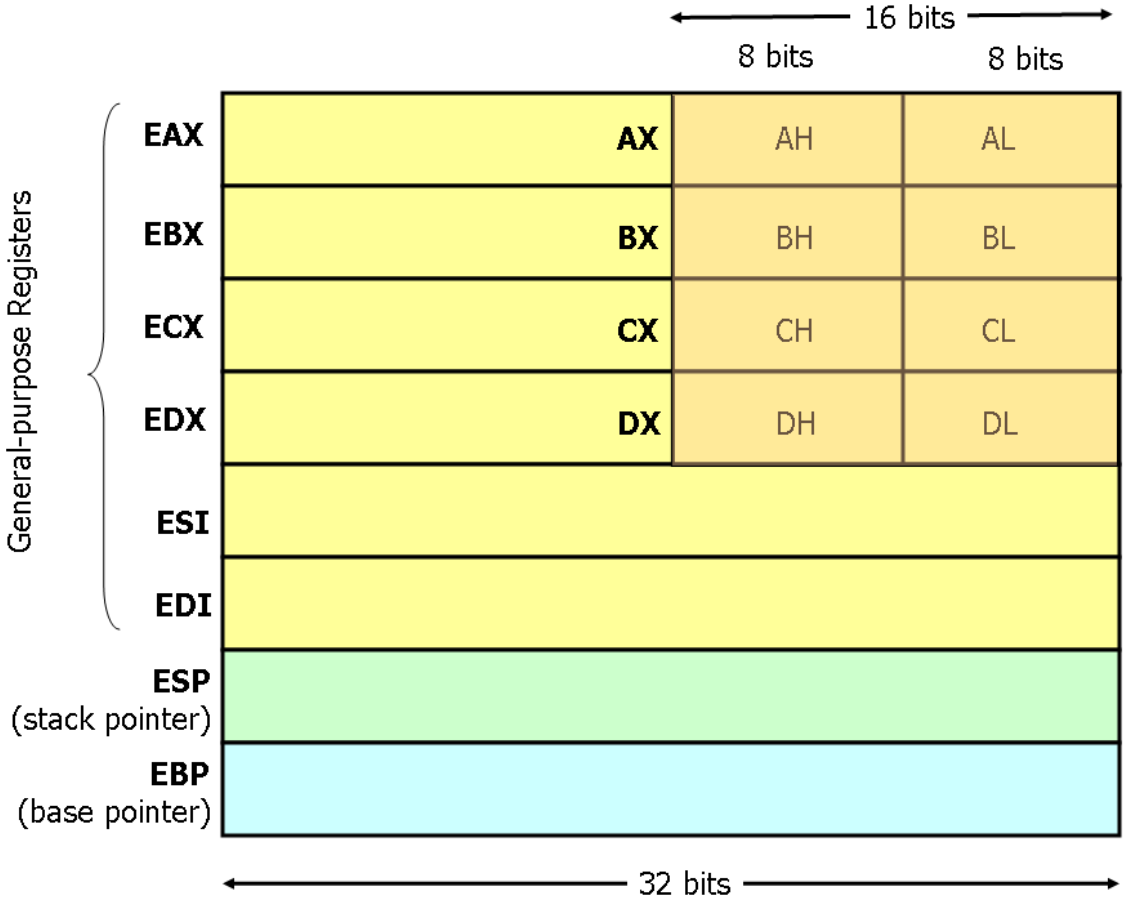
- To recompile xv6 to include the new command use the Linux shell and from the **xv6-public** folder run:

```
$ sudo make clean
$ sudo make qemu-nox
```

- From the **VM/xv6** shell, you can test the new command by running **hello** from the command line.

# x86 General Purpose Registers

- x86 is a **CISC** ISA with memory operands for non-load/store instructions and more complex addressing modes than **RISC**.
- x86 processors have eight 32-bit general purpose registers.
- For the EAX, EBX, ECX, and EDX registers, subsections may be used. For example, the least significant 2 bytes of EAX can be treated as a 16-bit register called AX. The least significant byte of AX can be used as a single 8-bit register called AL.



# x86 Data Transfer

- Between registers:

```
movb   %al, %ah   ; %ah = %al (1st operand is source and 2nd is destination)
movw   %ax, %bx   ; %bx = %ax
movl   %eax,%ebx  ; %ebx = %eax
```

- To/from memory locations:

```
movl %eax, (%ebp)      ; Store to memory: mem[%ebp] = %eax
movl %eax, 8(%ebp)     ; Store to memory: mem[%ebp + 8] = %eax
movl -4(%ebp), eax     ; Load from memory: %eax = mem[%ebp – 4]
```

- Constants to registers:

```
movl $10, %eax         ; %eax = 10
movw $0x1000, %ax      ; %ax = 4096 (decimal value of 0x1000)
movb $'A', %ah         ; %ah = 65 (A's ASCII code)
```


# xv6 Boot Loader

- The boot loader is a small software routine executed as soon as the processor is booted (in our case as soon as the QEMU VM starts).
- The processor is directed (e.g., through executing the system BIOS instructions) to run the boot loader software from the **master boot record** of a permanent storage unit, such as a hard drive or external USB drive.
- The boot loader is **in charge of loading** the code for the operating system from the permanent storage unit to the physical memory starting at address 0x100000.
- The xv6 code loaded at this address is originally written in x86 assembly language and can be found in file `entry.S`


# xv6 entry.S

- `entry.S` is an x86 assembly code program that sets different processor registers to configure the **memory paging** structure and **kernel stack** used by xv6.
- The code at end the of `entry.S` jumps to function `main()` (originally written in `main.c`) and starts executing the remaining code of the xv6 kernel.
- Most of the xv6 kernel source code is written in C while some code is written in x86 assembly. However, the code loaded to the memory for execution is always the executable machine code (compiled or assembled) of those source codes.


# xv6 main.c

- Function `main()` carries out the following bootstrap tasks:
  1. Calling function `tvinit()` (defined in `trap.c`) to initialize the trap vectors array where each `vectors[i]` contains the address of the code to handle interrupt `i`. The `vectors` array itself is defined in `vectors.S`. A special entry is `vectors[64]` that handles the system calls interrupt.
  2. Starting the first user process by calling `userinit()`, defined in `proc.c`, which is responsible for setting up the environment that the user sees including the xv6 shell.
  3. Calling function `mpmain()` that starts the scheduler by calling function `scheduler()`. Function `scheduler()` never returns (it goes into an infinite loop).

*Note:* Running `ps` right after logging to Linux will show the shell process (e.g., `bash`) and the `ps` process itself.


# System Calls

- Running a process in xv6 involves execution of **user program code** and **system code** (through **system (environment) calls/ trap instructions/interrupts**).
- The processor runs the user code in **user mode** (low privilege) while it runs the system code in **kernel mode** (high privilege).
  - A "**mode bit**" in the processor distinguishes whether the processor is running user code or kernel code.
  - In Kerner mode, the processor allows more **special instructions** to be executed and **restricted memory regions** to be accessed.
  - A **protection bit** for memory page entries distinguishes whether a memory page is for user instructions/data or for kernel instructions/data.
- **System (Environment) calls** allow user level processes to request services from the kernel. A way to control how **critical resources** are accessed/shared among multiple processes.


# Kernel Mode Cases

- There are three cases when control must be transferred from a user program to the kernel mode.

1. **System calls**: when a user program asks for an operating system service.
2. **Software exceptions**: when a program performs an illegal action.
   - Examples of illegal actions include divide by zero, attempt to access memory for a page-table entry that is invalid or not present, etc.
3. **Hardware interrupts**: when a device generates a signal to indicate that it needs attention from the operating system.
   - For example, a **clock chip** may generate an interrupt every 100 msec to allow the kernel to implement CPU virtualization through its scheduler.
   - As another example, when a block of data is read from **the disk**, it generates an interrupt to alert the operating system that the block is ready to be retrieved.


# xv6/x86 Interrupts Handling

- The x86 allows for 256 different interrupts to handle all the previous three kernel mode cases .

- xv6 maps them as follows:
  - Interrupts 0-31 are defined for **software exceptions**.
  - Interrupts 32-63 are mapped to the 32 **hardware interrupts**.
  - It uses interrupt 64 for the **system calls** interrupt.

- xv6 doesn't allow user processes to raise interrupts like device interrupts with the `int` instruction. If they try, they will encounter a general protection exception.


# xv6 Hardware Interrupts

- xv6 is designed for a system with **multiple processors**.
- For example, xv6 routes **keyboard interrupts** to the first processor (processor 0) and routes **disk interrupts** to the highest numbered processor on the system.
- Each processor can receive **timer interrupts** independently from the timer chip. xv6 sets it up in `lapicinit()` function defined in `lapic.c`
- The instruction `cli` disables interrupts on a processor by clearing `IF` (*Interrupt Flag*), and `sti` enables them.
- The scheduler on each processor enables interrupts. However, to control that certain code fragments during scheduling are not interrupted, xv6 disables interrupts during these code fragments (e.g., `switchuvm`, which is called to switch the process's address space).


# Interrupts and Device Drivers (1 of 3)

- A *driver* is the code in an operating system that manages a particular device: it tells the device hardware to perform operations, configures the device to generate interrupts when done, and handles the resulting interrupts.
- In many operating systems, the drivers of all the connected devices account for more code in the operating system than the core kernel.
- Some drivers dynamically switch between **polling** and **interrupts**, because using interrupts can be expensive, but using polling can introduce delay until the driver processes an event.


# Interrupts and Device Drivers (2 of 3)

- **Polling** requires the CPU (through the device driver code) to poll the device repeatedly for an I/O completion.
- In many computer architectures, the CPU execute simple instructions to poll a device: **read a device status register**, **logical-and to extract a status bit**, and **issue a system call if not zero**.
- Polling becomes inefficient when it is attempted repeatedly yet rarely finds a device ready for service.
- In such instances, it may be more efficient to arrange for the hardware controller to notify the CPU when the device becomes ready for service, which is the idea of **Interrupts**.


# Interrupts and Device Drivers (3 of 3)

- A network driver that receives a burst of packets may switch from interrupts to polling since it knows that more packets must be processed, and it is less expensive to process them using polling. Once no more packets need to be processed, the driver code may switch back to interrupts, so that it will be alerted immediately when a new packet arrives.
- Furthermore, a network driver might arrange to deliver interrupts for packets of one network connection to the processor that is managing that connection, while interrupts for packets of another connection are delivered to another processor.
  - This routing can get quite sophisticated; for example, if some network connections are short lived while others are long lived, and the operating system wants to keep all processors busy to achieve high throughput.

# x86 System Calls

- xv6 supports system calls through **software interrupt** instruction `int $64`
- This interrupt instruction expects the following before being invoked:
  - Register `%eax` contains the system call number
  - Registers `%ebx`, `%ecx`, `%edx`, `%esi`, `%edi`, `%ebp` contain the optional arguments (e.g., a system call that writes a string on the screen, its arguments are the address of the string to be written to the console and its length).
- The interrupt instruction optionally returns a value in register `%eax`
- *Example*: the following code is for the system call exit that terminates a process with exit code 0:

```
movl $2, %eax          ; 2 for SYS_exit
movl $0, %ebx          ; code = 0
int  $64
```


# General System Call Timeline

- The interrupt instruction stops the execution of the user process instructions and starts executing a new sequence of instructions called an **interrupt handler**.
- Before starting the interrupt handler, the processor **saves its registers**, so that the operating system can restore them when it returns from the handler.
- A challenge in the transition to and from the interrupt handler is that the processor should switch from **user mode** to **kernel mode**, and back.
- The terms **trap** and **interrupt** are usually used interchangeably.


# System Calls vs User Library Functions

- To a programmer, a system call looks like any other procedure call to a library function.
- If performance is an issue and if a task can be accomplished without a system call the program will run faster. Why?
  - Every system call involves overhead time in switching from the **user mode** to the **kernel mode**, and then switching back.
  - The operating system may **schedule another process** to run when a system call completes, further slowing the execution of a calling process.
- Another programming tip is to minimize the number of system calls.
  - For example, instead of using `printf` in a loop, you can combine the output in a string and print that string once outside the loop.


# xv6 Supported System Calls (1 of 2)

| | |
|---|---|
| `fork()` | Create a process. |
| `exit()` | Terminate current process. |
| `wait()` | Wait for child process to exit. |
| `pipe(fds)` | Create a pipe and return file descriptors to it. |
| `read(fd, buf, n)` | Read *n* bytes from an open file into *buf*. |
| `kill(pid)` | Terminate process *pid*. |
| `exec(filename, argv)` | Load a file and execute it. |
| `fstat(fd)` | Return information about an open file. |
| `chdir(dirname)` | Change the current working directory. |
| `dup(fd)` | Duplicate file descriptor. |
| `getpid()` | Return current process's id. |
| `sbrk(n)` | Grow process's memory by *n* bytes. |
| `sleep(n)` | Sleep for *n* seconds. |


# xv6 Supported System Calls (2 of 2)

| | |
|---|---|
| `uptime()` | Return number of clock tick interrupts since start. |
| `open(filename, flags)` | Open a file, *flags* indicate read/write. |
| `write(fd, buf, n)` | Write *n* bytes to an open file. |
| `mknod(name, major, minor)` | Create a device file. |
| `unlink(filename)` | Remove a file. |
| `link(src, dest)` | Create a hard link for a file. |
| `mkdir(dirname)` | Create a directory. |
| `close(fd)` | Close a file. |

- Each of the above system calls has a unique system call number.
- xv6 needs to use register `%eax` to pass this number to `int $64`.
- Notice that many system calls require passing one or more parameters.
- Does xv6 use other registers to pass those parameters?

# xv6 System Call Numbers

- The xv6 system call numbers are defined in `syscall.h` as shown.
- These are the numbers that need to be loaded in register `%eax` before calling `int $64`

```
// System call numbers
#define SYS_fork    1
#define SYS_exit    2
#define SYS_wait    3
#define SYS_pipe    4
#define SYS_read    5
#define SYS_kill    6
#define SYS_exec    7
#define SYS_fstat   8
#define SYS_chdir   9
#define SYS_dup    10
#define SYS_getpid 11
#define SYS_sbrk   12
#define SYS_sleep  13
#define SYS_uptime 14
#define SYS_open   15
#define SYS_write  16
#define SYS_mknod  17
#define SYS_unlink 18
#define SYS_link   19
#define SYS_mkdir  20
#define SYS_close  21
```

# xv6 System Call Definitions

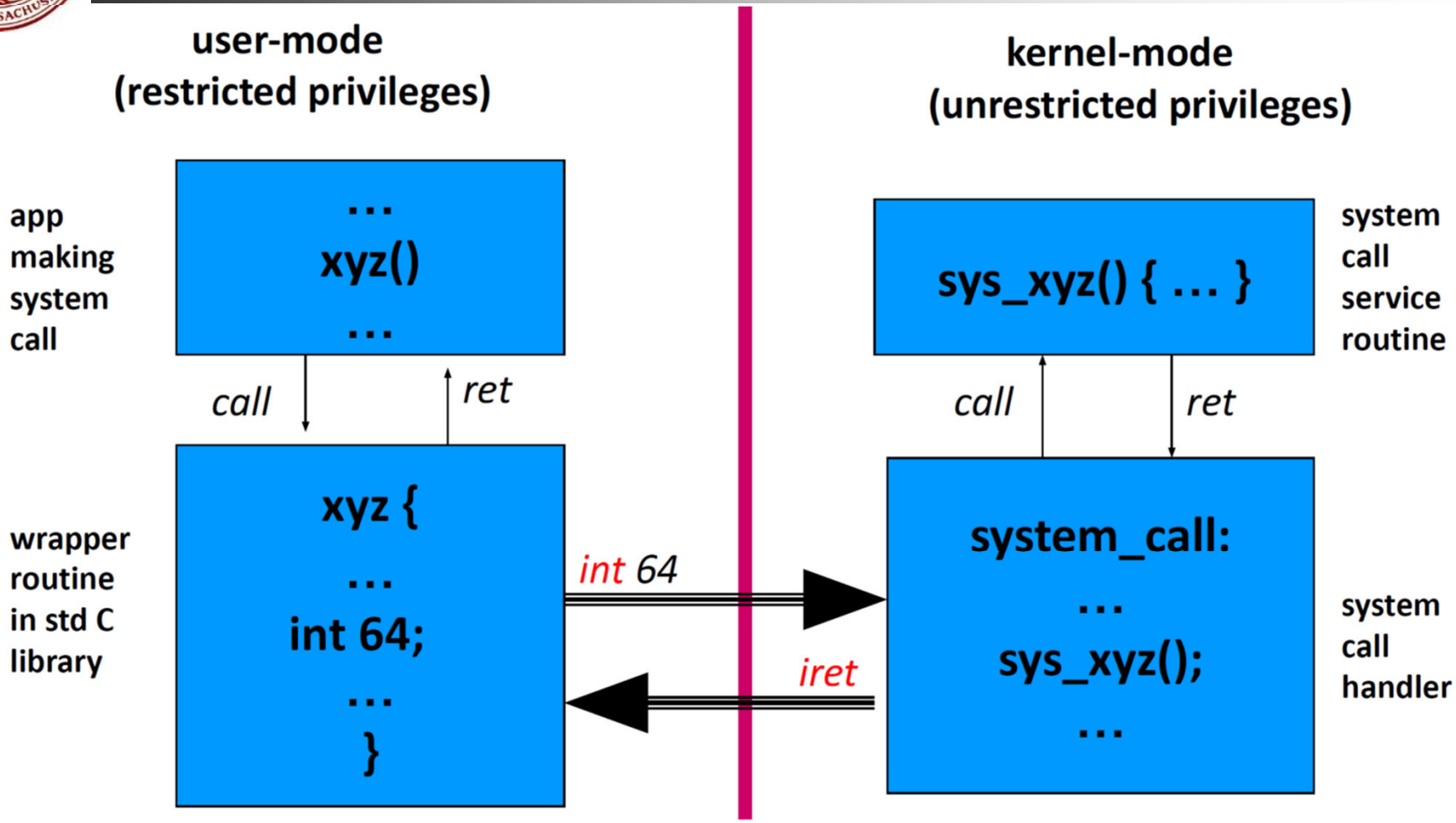
- xv6 uses the shown macro in `usys.S` to generate the `eax` initialization and `int $64` instructions.
- To store it in register `eax`, the interrupt number is retrieved from `syscall.h` using `SYS_ ## name` where ## is the merging operator.
- The system call is issued using `int $T_SYSCALL` (`$T_SYSCALL` is defined as 64 in `traps.h`)
- Every line in the code body, such as `SYSCALL(fork)`, will be replaced with the macro expansion.
- **xv6 does not use arguments** for the `int $64` calls as it keeps the arguments to those calls in the **user stack** and grab them later.

```
#include "syscall.h"
#include "traps.h"

#define SYSCALL(name) \
  .globl name; \
  name: \
    movl $SYS_ ## name, %eax; \
    int $T_SYSCALL; \
    ret

SYSCALL(fork)
SYSCALL(exit)
SYSCALL(wait)
SYSCALL(pipe)
SYSCALL(read)
SYSCALL(write)
SYSCALL(close)
SYSCALL(kill)
SYSCALL(exec)
SYSCALL(open)
SYSCALL(mknod)
…
…
```

# User Invocation of a System Call in xv6




- The wrapper routine in *C* is needed so that user programmers do not need to deal with the details of settings the registers for the system call (int $64).
- `sys_xyz()` service routine usually calls another function called `xyz()` to complete the "service", however this kernel-mode `xyz()` routine is not the same as the user-mode `xyz()` wrapper routine.

# xv6 Process Structure

- The `struct proc` (in `proc.h`) is the per process data structure:

```
enum procstate // all possible process states
        { UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNING, ZOMBIE };
// Per-process data structure
struct proc {
  uint sz;                     // Size of process memory (bytes)
  pde_t* pgdir;                // Page directory
  char *kstack;                // Bottom of kernel stack for this process
  enum procstate state;        // Process state
  int pid;                     // Process ID
  struct proc *parent;         // Parent process
  struct trapframe *tf;        // Trap frame for current syscall
  struct context *context;     // defined in proc.h with registers needed for swtch()
  void *chan;                  // If non-zero, sleeping on chan
  int killed;                  // If non-zero, have been killed
  struct file *ofile[NOFILE];  // Open files
  struct inode *cwd;           // Current directory
  char name[16];               // Process name (debugging)
};
```

# xv6 Process Table

- The process table **ptable** (in **proc.c**) is implemented as an array of elements of type **struct proc**.
- Constant **NPROC** (in **param.h**) is set to 64, and limits the maximum number of active processes in xv6:

```
struct {
  struct spinlock lock;
  struct proc proc[NPROC];
} ptable;
```

- The entries in the process table are initialized by setting their **state** field to the value **UNUSED** (representing records that are not associated with an active process).
- The **lock** field in **ptable** is needed in case of multiple processors so that only one processor can acquire the lock to avoid concurrent access to the same table.

# Function Pointer in C

- We usually use C pointers to point at data stored in the memory.
- A pointer in C can also be used to point at the beginning of a function instructions in memory.
- *Example:*

```
#include <stdio.h>
float avg(int x, int y) {  /* function to find the average of two numbers */
  int s;       float a;
  s = x+y;     a = s/2.0;
 return a;
}

int main() {
  float (*avgP)(int, int);  /* function pointer declaration */
  avgP = avg;               /* pointer assignment */
  float av = avgP(10, 15);  /* function call using its pointer*/
  printf("Average = %f\n", av);
 return 0;
}
```

# Array of Function Pointers in C

- An array of function pointers can play a switch statement role for making a decision, as in this example:

```
#include <stdio.h>
float avg(int x, int y) { return (x+y)/2.0; }
float max(int x, int y) { return (x>y)?x:y; }
float min(int x, int y) { return (x<y)?x:y; }

int main() {
   // array of function pointers declaration and initialization
   float (*afp[3])(int, int) = {avg, max, min}; // size 3 is optional here
   for(int i =0; i<3; i++)
     printf("Function %d result = %f\n", i, afp[i](10, 15));
 return 0; }
```

- You can initialize the array elements in the above code out of order by specifying which array indices to initialize. Example:

```
float (*afp[3])(int, int) = {[2] avg, [0] max, [1] min};
```

Here the above loop will call max, min, and avg in this order.

# xv6 System Call Handler Table

- To locate the system call handler function in the xv6 kernel, a system call table data structure is used.
- The xv6 system call table is an array of function pointers, as shown, and is defined in **`syscall.c`**
- The indices of this table is the system call number. *Can we omit the explicit definition of the indices?*
- Notice all functions have **no parameters**. Doesn't a function like sys_kill require one parameter (pid)?
- The actual implementation of the system call handlers are not in this file. Therefore, multiple statements like the following need to be added:

```
extern int sys_exit(void);
extern int sys_fork(void);
```

```
static int (*syscalls[])(void)
= {
 [SYS_fork]    sys_fork,
 [SYS_exit]    sys_exit,
 [SYS_wait]    sys_wait,
 [SYS_pipe]    sys_pipe,
 [SYS_read]    sys_read,
 [SYS_kill]    sys_kill,
 [SYS_exec]    sys_exec,
 [SYS_fstat]   sys_fstat,
 [SYS_chdir]   sys_chdir,
 [SYS_dup]     sys_dup,
 [SYS_getpid]  sys_getpid,
 [SYS_sbrk]    sys_sbrk,
 [SYS_sleep]   sys_sleep,
 [SYS_uptime]  sys_uptime,
 [SYS_open]    sys_open,
 [SYS_write]   sys_write,
 [SYS_mknod]   sys_mknod,
 [SYS_unlink]  sys_unlink,
 [SYS_link]    sys_link,
 [SYS_mkdir]   sys_mkdir,
 [SYS_close]   sys_close,
};
```

# Handling int $64

- The `int $64` instruction is handled by `vector64` in `vectors.S` as shown:

```
vector64:
  pushl $0
  pushl $64
  jmp alltraps
. . . . .
.globl alltraps
alltraps:
. . . . .
# Call trap(tf), where tf=%esp
  pushl %esp
  call trap
. . . . .
void trap(struct trapframe *tf){
  if(tf->trapno == T_SYSCALL){
    if(myproc()->killed) exit();
    myproc()->tf = tf;
    syscall();
    if(myproc()->killed) exit();
    return;    }
```

- The handler jumps to `alltraps` in `trapasm.S`, which pushes in the stack the needed registers values and calls `trap` function, in `trap.c`, with the address of the `trapframe` struct in the stack (`struct trapframe` is defined in `x86.h`)
- The shown code segment in the `trap` function stores the process trap frame and then calls the `syscall()` function in `syscall.c`

*Note*: The function `myproc()` is defined in `proc.c` and returns a pointer to the current CPU's `struct proc` (which is defined in `proc.h`).

# xv6 syscall() Function

- The following is the `syscall()` function in `syscall.c`

```
void syscall(void) {
  int num;
  struct proc *curproc = myproc();
  num = curproc->tf->eax;
  if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
    curproc->tf->eax = syscalls[num]();                    }
  } else                                                   {
    cprintf("%d %s: unknown sys call %d\n",
            curproc->pid, curproc->name, num);
    curproc->tf->eax = -1;                                 }
}
```

- The function loads the system call number from the trap frame, which contains the saved `%eax`.
- If `num` is a valid number (between 0 and max number of `syscalls`) and if its service routine in the `syscalls` table is not null, then call that service routine.
- The return value of this function is stored back in register `eax` in the user process trap frame.


# System Calls Arguments

- xv6 does not use arguments for the `int $64` system call as all functions in the handler table must have the same signature. Instead, it keeps the arguments to those calls in the user stack and grab them using one of the following **helper functions** defined in `syscall.c`.
- All functions start with argument *n* for the function to fetch the ***n*th** argument of the system call from the user stack.
  - `argint(int n, int *ip)` fetches a 32-bit argument as integer.
  - `argptr(int n, char **pp, int size)` fetches the argument and checks that this argument is a valid user-space pointer.
  - `argstr(int n, char **pp)` interprets the argument as a pointer and ensures that the pointer points at a NULL-terminated string and that the complete string is located within the user part of the address space.
  - `argfd(int n, int *pfd, struct file **pf)` is defined in `sysfile.c` and uses `argint` to retrieve a file descriptor number, checks if it is valid file descriptor, and returns the corresponding struct file.

# xv6 System Call Handler Implementation

- The actual handling of different system calls are implemented in **`sysproc.c`** where you can find the implementation of functions such as the shown **`sys_kill`**

```
int sys_kill(void) {
  int pid;
  if(argint(0, &pid) < 0)
    return -1;
  return kill(pid);
}
```

- The body of the function reads its arguments from the user stack using the helper function **`argint`**, and not from the currently active kernel stack. Therefore, the function implementing the system call takes no explicit arguments in its header.
- *Note*: the **`sys_kill()`** function calls the **`kill()`** function, which is the implementation of killing a process and defined in **`proc.c`**

# The kill() Function Implementation

- The main task of this function is to set the `killed` attribute of process `pid` to 1 and, if it is sleeping, wakes it up.
- However, that killed process won't **exit** until it enters or leave the kernel by making a system call or because of the timer (or some other device) interrupts.
- Recall that interrupt handler calls `trap` in `trap.c`, which after returning from the `syscall()` call, it checks for the `killed` attribute and if it is 1, it exits the process.

```
// Kill the process with the given pid.
// Process won't exit until it returns to user space
//          (refer to trap in trap.c).
int kill(int pid) {
  struct proc *p;
  acquire(&ptable.lock);
  // search the  proc table for PID match
  for(p = ptable.proc;  //proc is the processes array
      p < &ptable.proc[NPROC]; p++){
    if(p->pid == pid){
      p->killed = 1;
      // Wake process from sleep if necessary.
      if(p->state == SLEEPING)
        p->state = RUNNABLE;
      release(&ptable.lock);
      return 0;
    }
  }
  release(&ptable.lock);
  return -1;
}
```

# exit() Vs. kill()

- While `exit()` allows a process to terminate itself, `kill()` lets one process request that another process (the victim) terminate.
- To terminate a process, `exit()` performs some tasks on the process including closes all its open files, sets its state to ZOMBIE, and wakes up its parent.
- It would be too complex for `kill()` to directly terminate the victim process, since the victim might be executing on another CPU, perhaps in the middle of a sensitive sequence of updates to kernel data structures.

# xv6 User Library

- The user application that wishes to make a system call (e.g., sleep()) calls the corresponding C function prototype declared in the shown "`user.h`" file.
- These are just prototypes used by the C compiler to pass the arguments (by pushing them in the stack), and there is **no C implementation** of them.
- Instead, the system will call the corresponding assembly code generated by the macro in `usys.S` (where the name of the shown functions are defined as "global")

```
// system calls
int fork(void);
int exit(void) __attribute__((noreturn));
int wait(void);
int pipe(int*);
int write(int, const void*, int);
int read(int, void*, int);
int close(int);
int kill(int);
int exec(char*, char**);
int open(const char*, int);
int mknod(const char*, short, short);
int unlink(const char*);
int fstat(int fd, struct stat*);
int link(const char*, const char*);
int mkdir(const char*);
int chdir(const char*);
int dup(int);
int getpid(void);
char* sbrk(int);
int sleep(int);
int uptime(void);
```

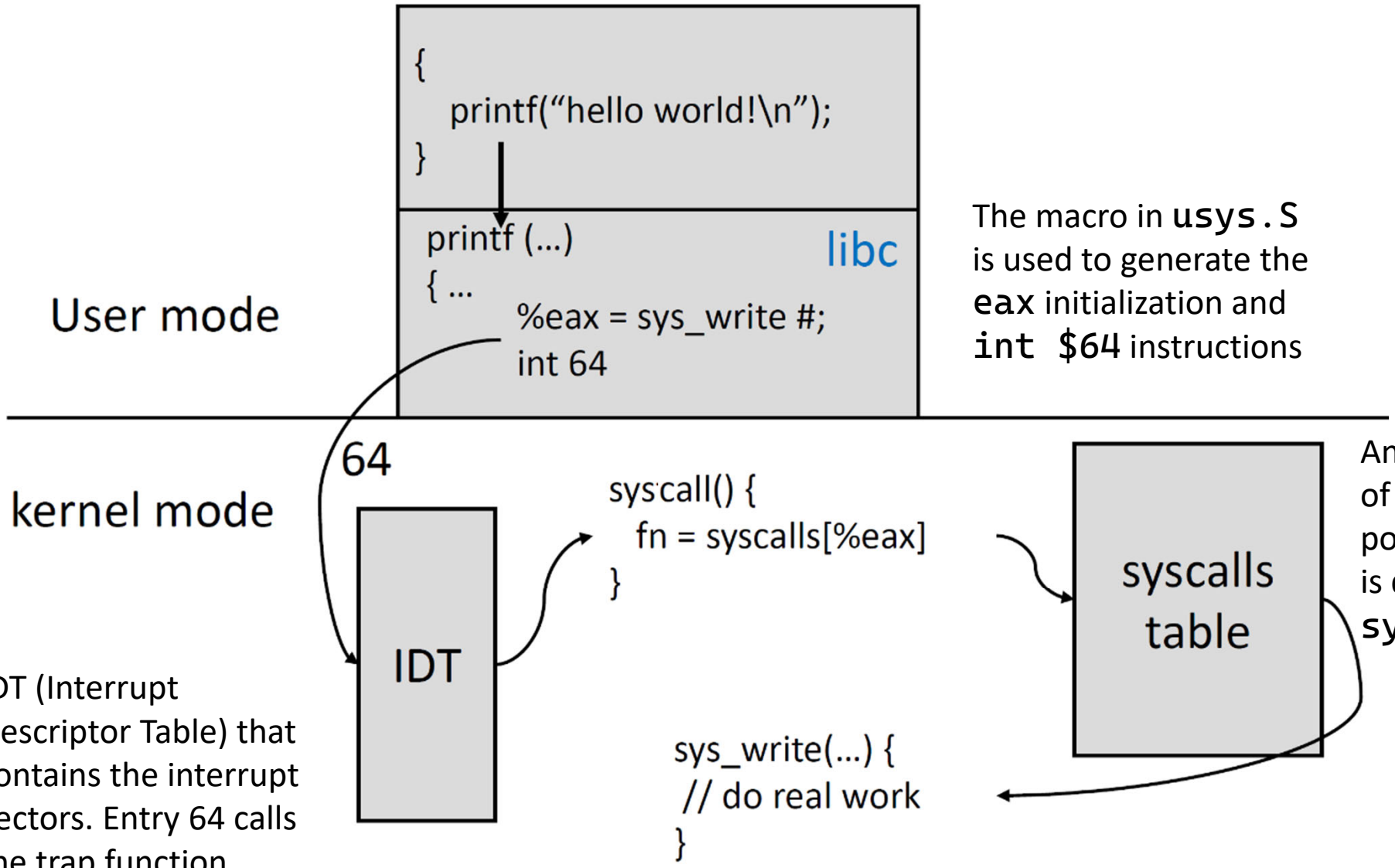

# Modifying `syscall()`

- In the following example, we will modify the trap handler for system calls in order to make xv6 print a message whenever any of the system calls **`fork`**, **`wait`**, or **`exit`** is invoked.
- While in the Linux shell, edit **`syscall.c`** and modify its **`syscall()`** function by adding the following red lines: (Note: in the kernel code, C function **`printf`** is not available. Instead, kernel code should invoke its own version of printf, called **`cprintf`** (prefix c stands for console)

```
if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
 if (num == SYS_fork) cprintf("-----> syscall fork\n");
 if (num == SYS_wait) cprintf("-----> syscall wait\n");
 if (num == SYS_exit) cprintf("-----> syscall exit\n");
 curproc->tf->eax = syscalls[num]();
} else {
```

- From the **`VM/xv6`** shell, you can now test different commands and track the time sequence of calling **`fork`**, **`wait`**, and **`exit`**.

# System Call Example: `printf()`




The macro in `usys.S` is used to generate the `eax` initialization and `int $64` instructions

An array of function pointers that is defined in `syscall.c`

IDT (Interrupt Descriptor Table) that contains the interrupt vectors. Entry 64 calls the trap function, which in turn calls syscall().

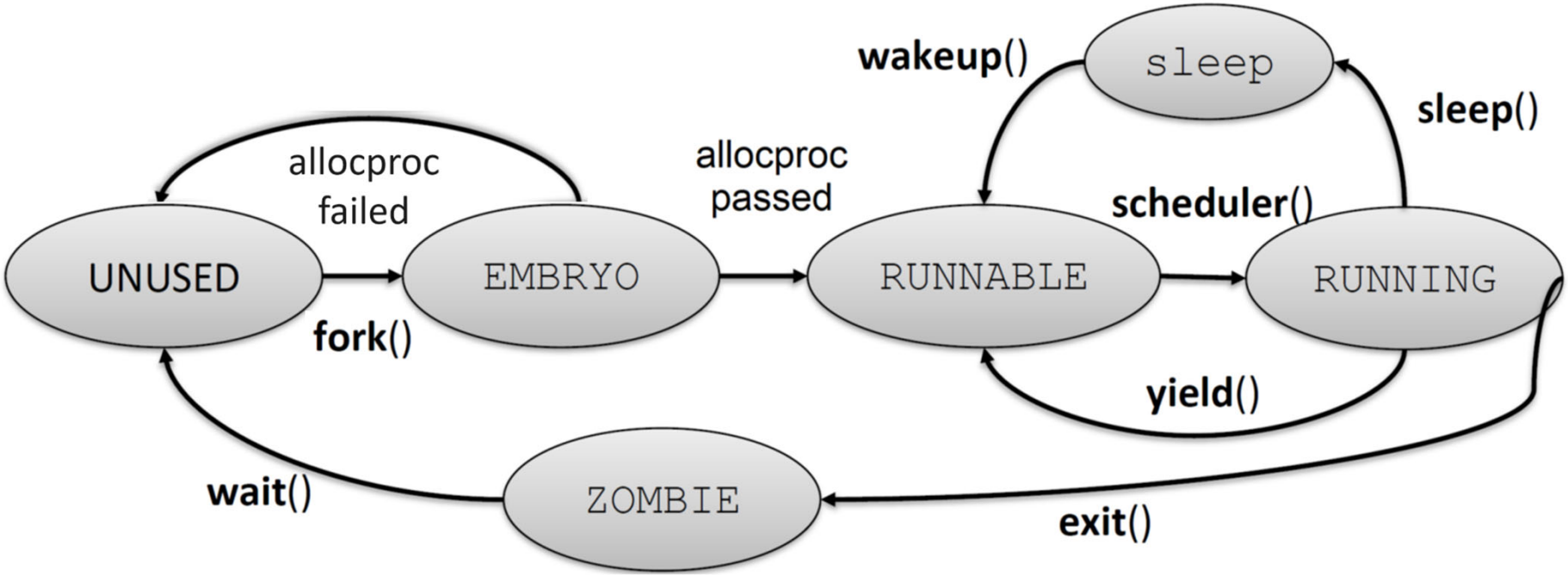
# Adding a System Call to xv6

- According to the xv6 system calls discussion so far, to add a new system call to xv6, you need to modify the following files:

1. syscall.h → to add a unique numeric identifier for the new system call. This number is used later for indexing in the array of function pointers. It is the number that needs to be loaded in register `%eax` before calling int `$64`
2. usys.S → to add a line for the new system call so that the needed instructions are generated by the macro in this file.
3. syscall.c → to add a new entry of the system call function (handler) to the `syscalls` table (array) of functions pointers. Also add in this file the "extern" statement of the system call function declaration.
4. sysproc.c → to add the implementation of that handler.
5. user.h → to add the user-level prototype of the new system call here.

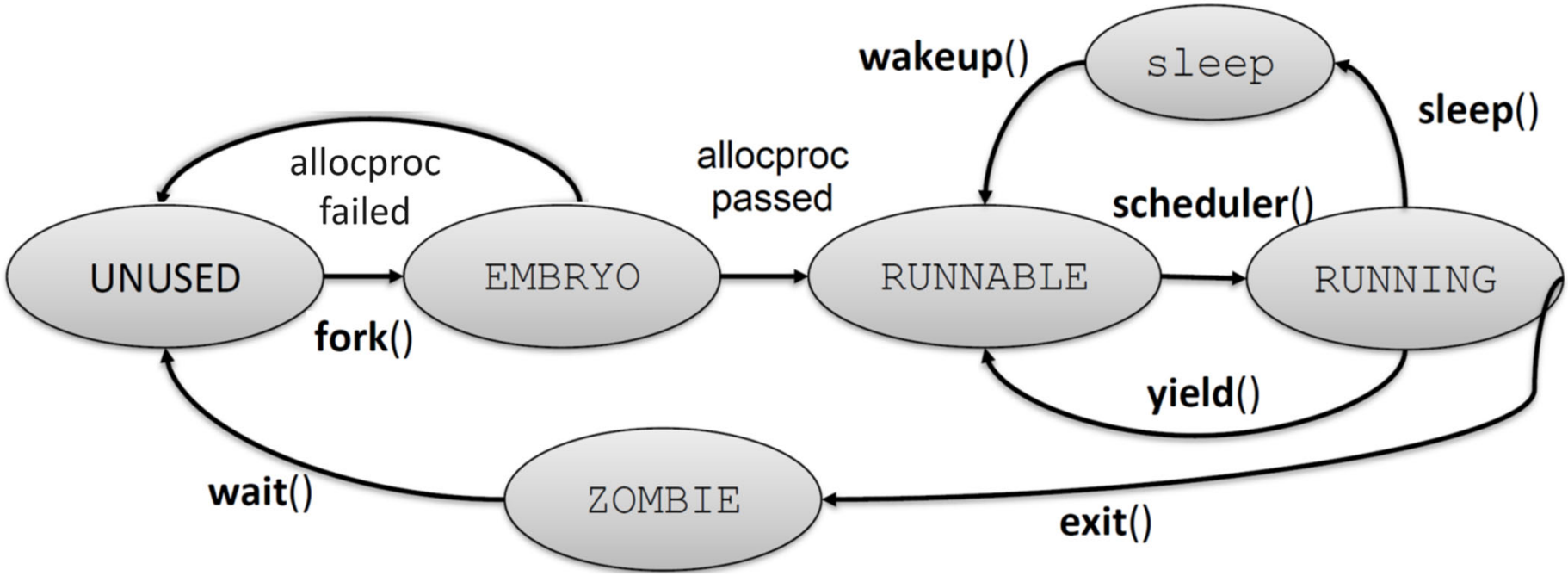
# xv6 Process Lifecycle (1 of 2)

- The figure summarizes the functions that trigger the state transition of a process in xv6. The functions are in **`proc.c`**
- A newly forked process is in the **EMBRYO** state. If the process memory allocation fails, **`allocproc`** changes the state of the record assigned to that process back to **UNUSED**, otherwise it sets it to **RUNNABLE** state.



# xv6 Process Lifecycle (2 of 2)

- When a process exits, it switches to the `ZOMBIE` state until the parent calls wait to learn of the exit. The parent is then responsible for freeing the memory associated with the process and changing its entry in the `ptable` to `UNUSED`.
  - A zombie process is the one that dies (i.e., finishes its execution) but its parent does not check on it via `wait()`.
  - An orphan process is the one that its parent dies.



# xv6 First User Process

- The first user process is the "`init`" process defined in `init.c`.
- To prepare for the `init` process, `userinit()` is in charge of allocating an `UNUSED` entry in the process table for it.
- It sets the `state` field of the `proc` struct of that entry to `RUNNABLE` so that it could be scheduled by the xv6 scheduler.

```
p->state = RUNNABLE;
```

- Once `userinit()` finishes execution, `init` is ready to run.
- Once the scheduler moves it to the `RUNNING` state, the `init` process forks a child process, which converts to the xv6 "shell".
- From the xv6 "shell" process, the user can "create" (fork + exec) more processes.
- The scheduler will handle execution of theses multiple processes.

# xv6 Scheduler (1 of 2)

- The following loop is in the **`scheduler`** function, defined in **`proc.c`**
- Scheduler never returns as it loops infinitely implementing a round-robin scheduling policy

```
for(;;){
 sti();  // Enable interrupts as xv6 disables interrupts on a CPU when it holds a lock
 acquire(&ptable.lock);  // to support multi-processor access to the table
 // Loop over process table looking for process to run
 for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
  if(p->state != RUNNABLE) continue;
  c->proc = p;  //setting the per-CPU current process variable proc
  switchuvm(p);  // switch to the process's page table (i.e., address space)
  p->state = RUNNING;
  swtch(&(cpu->scheduler), p->context);  //process p resumes running
  switchkvm();  // switch back to the kernel page table
  // Process is done running. It should have changed its p->state before coming back.
  c->proc = 0;
 }
 release(&ptable.lock);
}
```

# xv6 Scheduler (2 of 2)

- The xv6 scheduler starts the loop by checking all the processes in `ptable` looking for a process that is in the RUNNABLE state.
- But before doing that and to be safe (due to concurrent access in case of multiple processors), the scheduler needs to acquire the lock.

```
acquire(&ptable.lock);
for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->state != RUNNABLE) continue;
    c->proc = p;
    ……
```

- `c` is of type pointer to `struct cpu` which records, among other things, the process currently running on processor `c`.

# Processes Switching (1 of 3)

- The scheduler switches to the chosen process using:

```
c->proc = p;
switchuvm(p);  // switch to the process address space
p->state = RUNNING;
swtch(&(cpu->scheduler), p->context);//process p resumes running
switchkvm();  // switch back to the kernel address space
```

- The scheduler calls the **`swtch`** function (defined in **`swtch.S`**) to switch the current CPU state (context) to the state of the about to run process: `swtch(&(cpu->scheduler), p->context);`
- Note **the scheduler context does not belong to any process object** and hence the CPU context is used in the above switching.

# Processes Switching (2 of 3)

- The prototype of the **swtch** function (defined in **swtch.S**) is:
  ```
  void swtch(struct context** old, struct context* new);
  ```
- It saves the current registers on the stack, creating a struct context, and save its address in ***old**. Switch stacks to **new** and pop previously-saved registers.
- Both **cpu->scheduler** and **p->context** are of type **struct context** defined in **proc.h** as shown:
- Switching from one context to another basically involves saving the current (old) context registers and restoring previously saved registers of the new context.
- This includes register **eip** that has the address of instruction to resume from (So, after **swtch**, the cpu runs the chosen process)

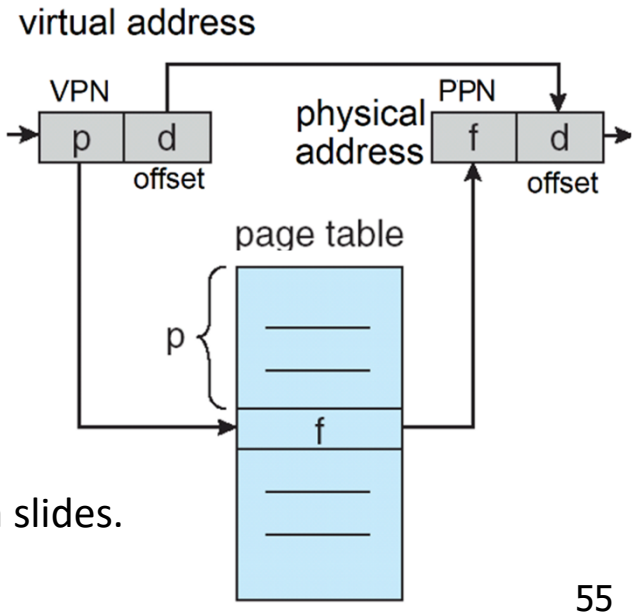
```
struct context {
  uint edi;
  uint esi;
  uint ebx;
  uint ebp;
  uint eip;
};
```


# Processes Switching (3 of 3)

- As a process is running (usually in the user mode) eventually it will enter the kernel mode due to one of the following events:
  - System calls
  - Software exceptions
  - Hardware interrupts (e.g., the timer interrupt)
- At the end of handling these events, the **`sched()`** function (defined in **`proc.c`**) is invoked, which in turn switches context from the current process to the scheduler context by calling : `swtch(&p->context, mycpu()->scheduler);`
- This will resume the scheduler from its `switchkvm()` code line.
- When it is time for the process to give up the CPU, the process does the following:
  - Changes its **`p->state`** to either **`SLEEPING`** or **`ZOMBIE`**.
  - Calls **`swtch`** (same way as above) to save its own context and return to the scheduler context to transfer control back to the scheduler.

# x86 Paging

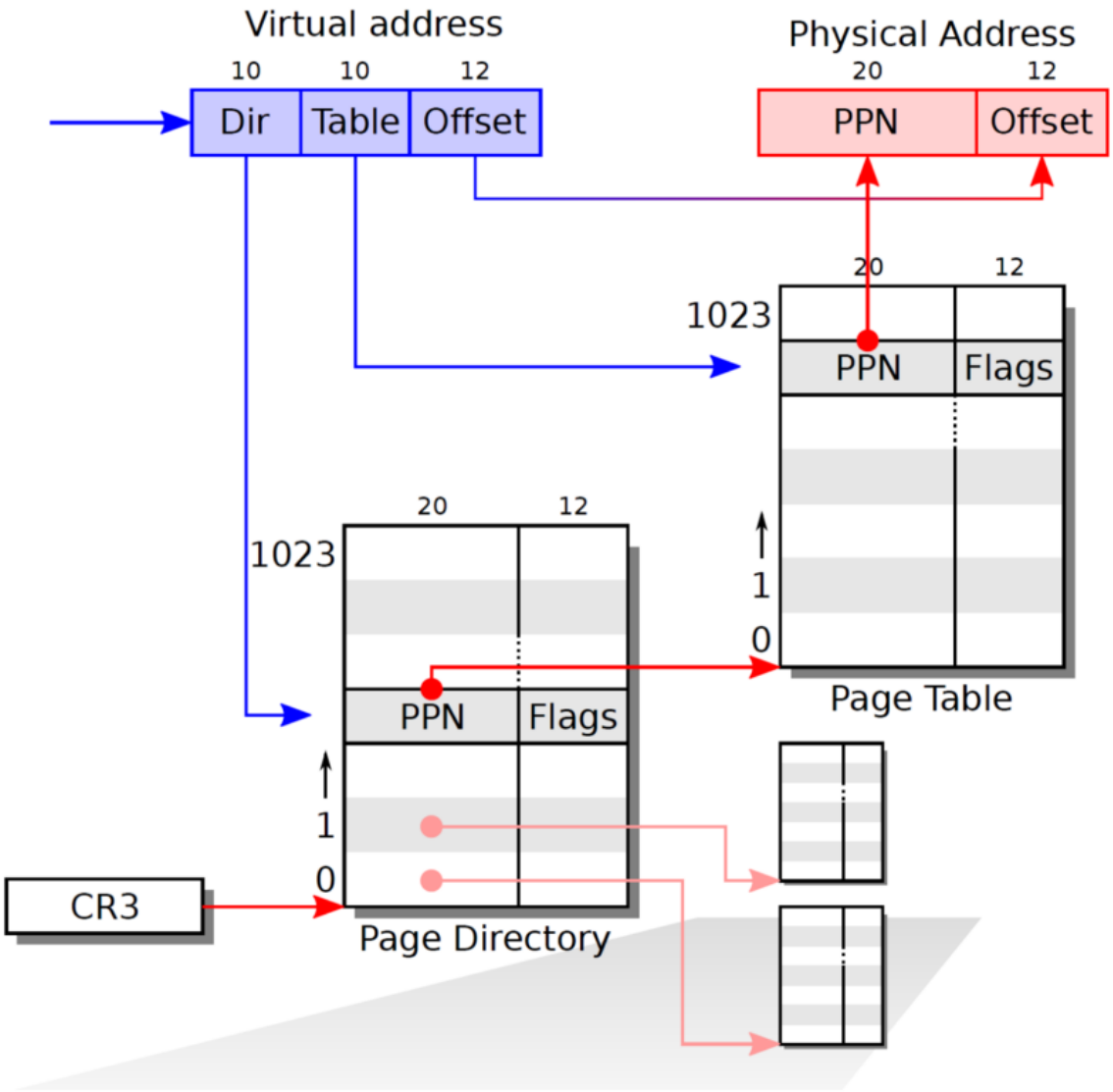
- x86 instructions (both user and kernel) manipulate virtual addresses.
- The x86 page table hardware maps each virtual address to a physical address.
- An x86 page table is logically an array of $2^{20}$ page table entries (**PTE**s). Each PTE contains a 20-bit physical page number (**PPN***) and some flags. The following are the flags defined in xv6 `mmu.h`
  - PTE_P indicates whether the PTE is **present**: if it is not set, a reference to the page causes a fault.
  - PTE_W controls whether instructions are allowed to issue writes to the page; if not set, only reads and instruction fetches are allowed.
  - PTE_U controls whether user programs are allowed to use the page; if clear, only the kernel is allowed to use the page.




* Referred to it as Physical Frame Number (PFN) in the Memory Virtualization slides.

# x86 Page Table Hardware

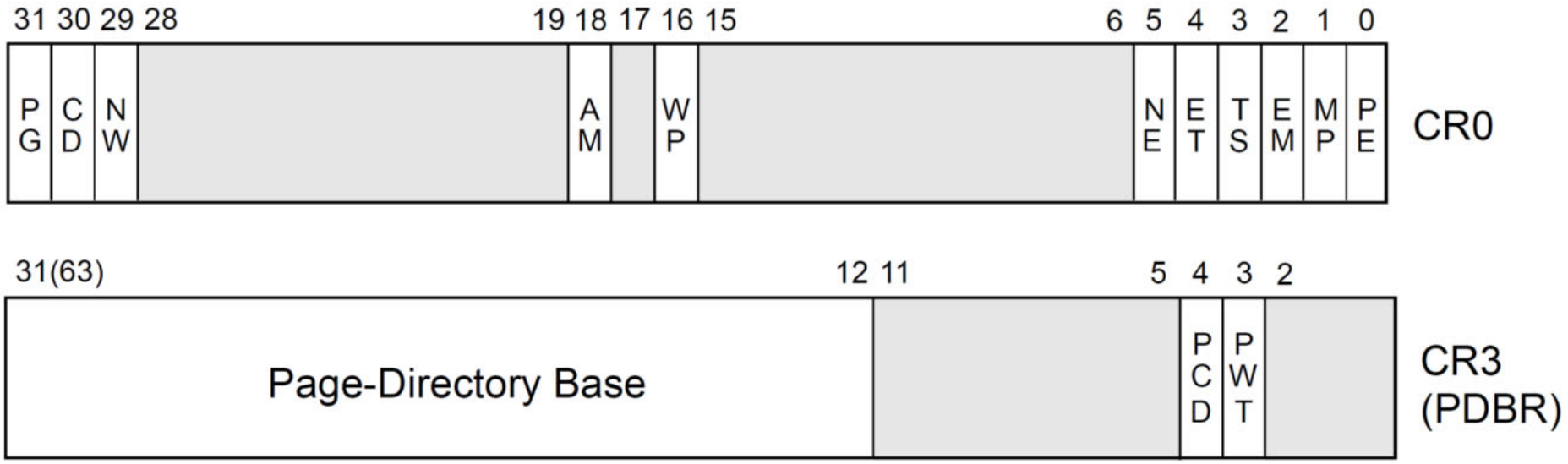
- As shown, a page table is stored in physical memory as a two-level tree.
- The root of the tree is a 4096-byte page directory that contains 1024 PTE-like references to page table pages.
- Each page table page is an array of 1024 32-bit PTEs.
- The most significant 20 bits of register `cr3` contains the page directory base address.
- The physical address of any entry in the directory is formed as: $[cr3]_{20}[Dir]_{10}[00]_2$




PPN: physical page number
PTE: page table entry

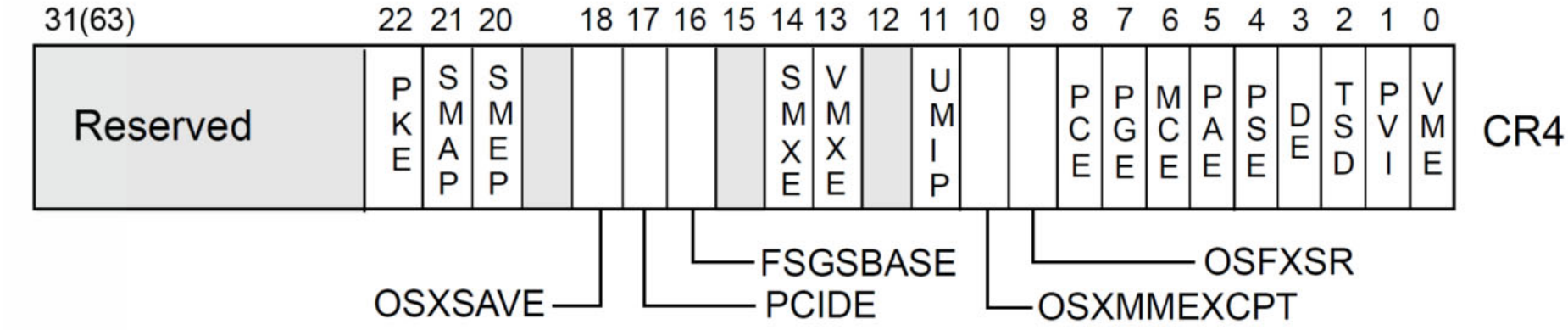
# x86 Paging Support Registers (1 of 2)

- Multiple registers and memory locations are involved in the address translation process in x86:
- Register **`%cr0`** contains bit **PG** at position 31.
  - When this bit is 1, paging is active, and **`%cr3`** contains the page directory base address in its 20 most significant bits.
  - When the **PG** bit is 0, paging is disabled, and the processor only uses physical addresses.



# x86 Paging Support Registers (2 of 2)

- Register `%cr4` contains bit PSE (Page Size Extension) at position 4. If this bit is set to 1, it activates support for 4MB pages, rather than the default 4KB page size.



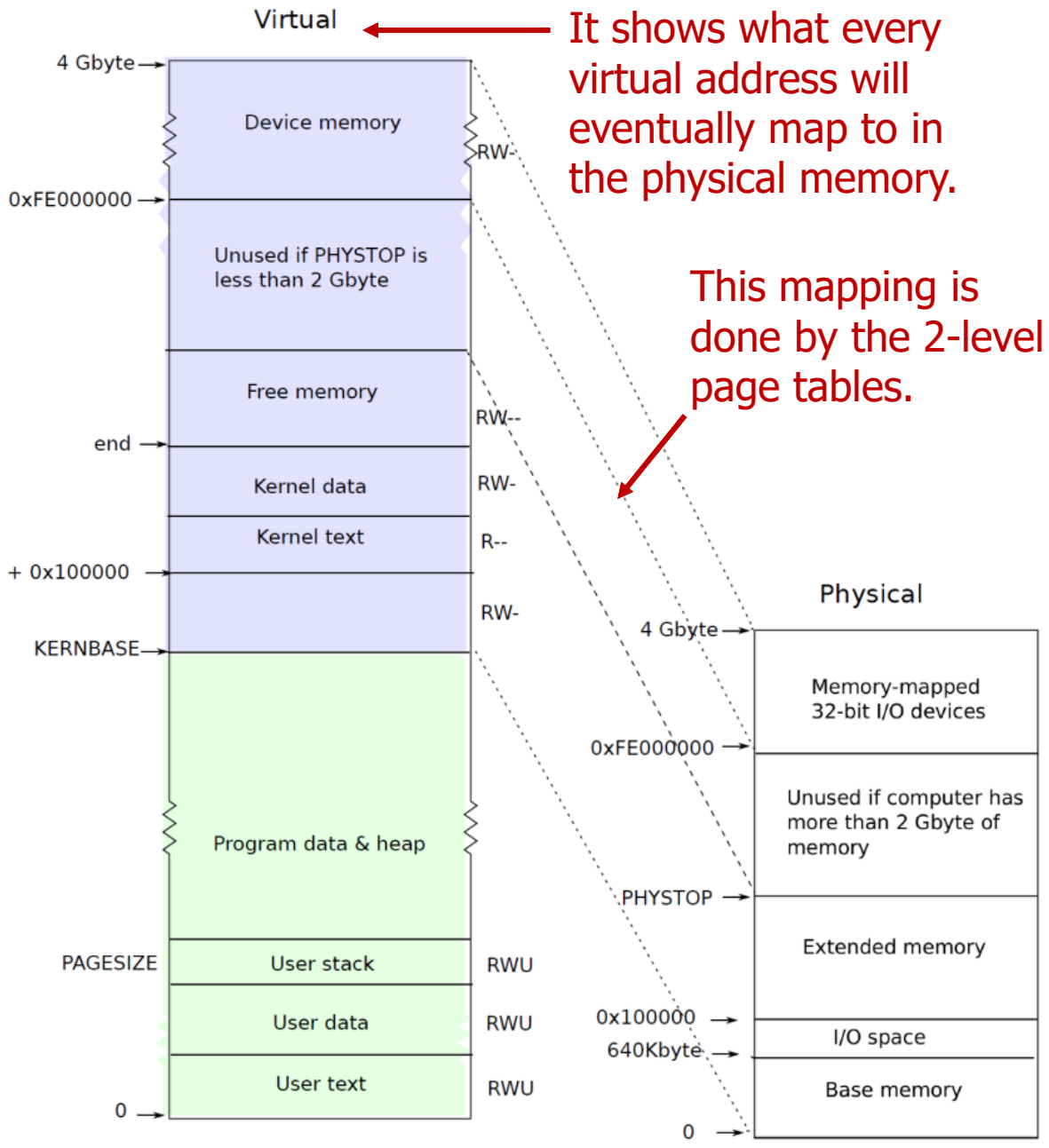
# xv6 and Page Table Hardware.

- xv6 uses 32-bit virtual addresses, resulting in a virtual address space of 4GB.
- xv6 uses paging to manage its memory allocations.
- xv6 uses a page size of 4KB, and a two-level page table structure.
- Before "running" a process, the scheduler stores the base address of the process's page table in the control register (`%cr3`).
- This is done in the xv6 scheduler, as shown below, by calling `switchuvm()`, which is defined in `vm.c`
- Function `walkpgdir()` in `vm.c` takes a virtual address and returns its corresponding PTE in the page table.

```
switchuvm(p);  // switch to the process's page table (i.e., address space)
p->state = RUNNING;
swtch(&(cpu->scheduler), p->context);  //process p resumes running
switchkvm();  // switch back to the kernel page table
```

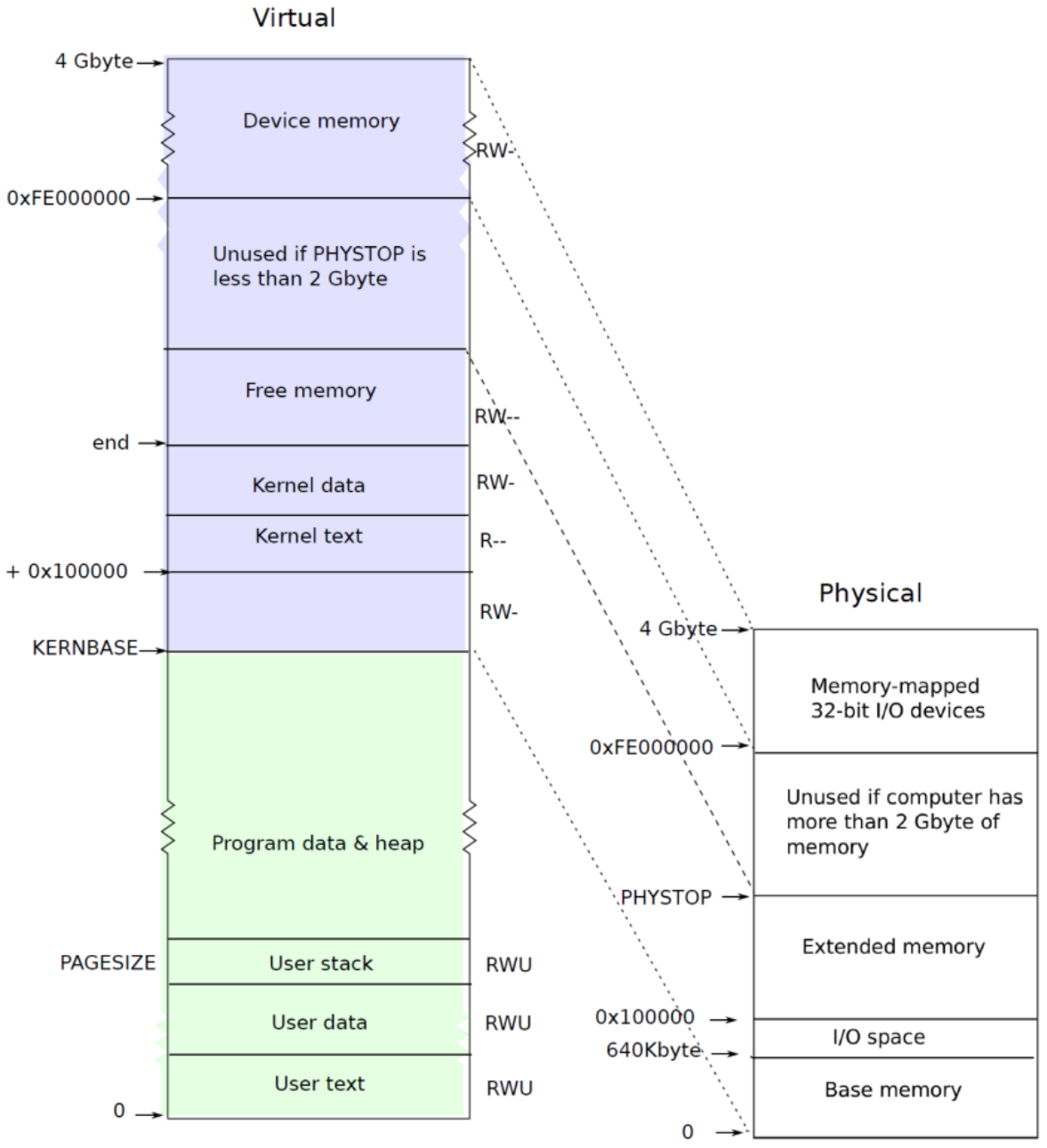
# xv6 Process Address Space (1 of 4)

- In xv6 each process has its own **page table**, and xv6 tells the page table hardware to switch page tables when xv6 switches between processes.
- As shown, a process's user memory starts at virtual address zero and can grow up to **KERNBASE** = 0x80000000, allowing a process to address up to **2 gigabytes** of memory (most processes do not use this entire user space).



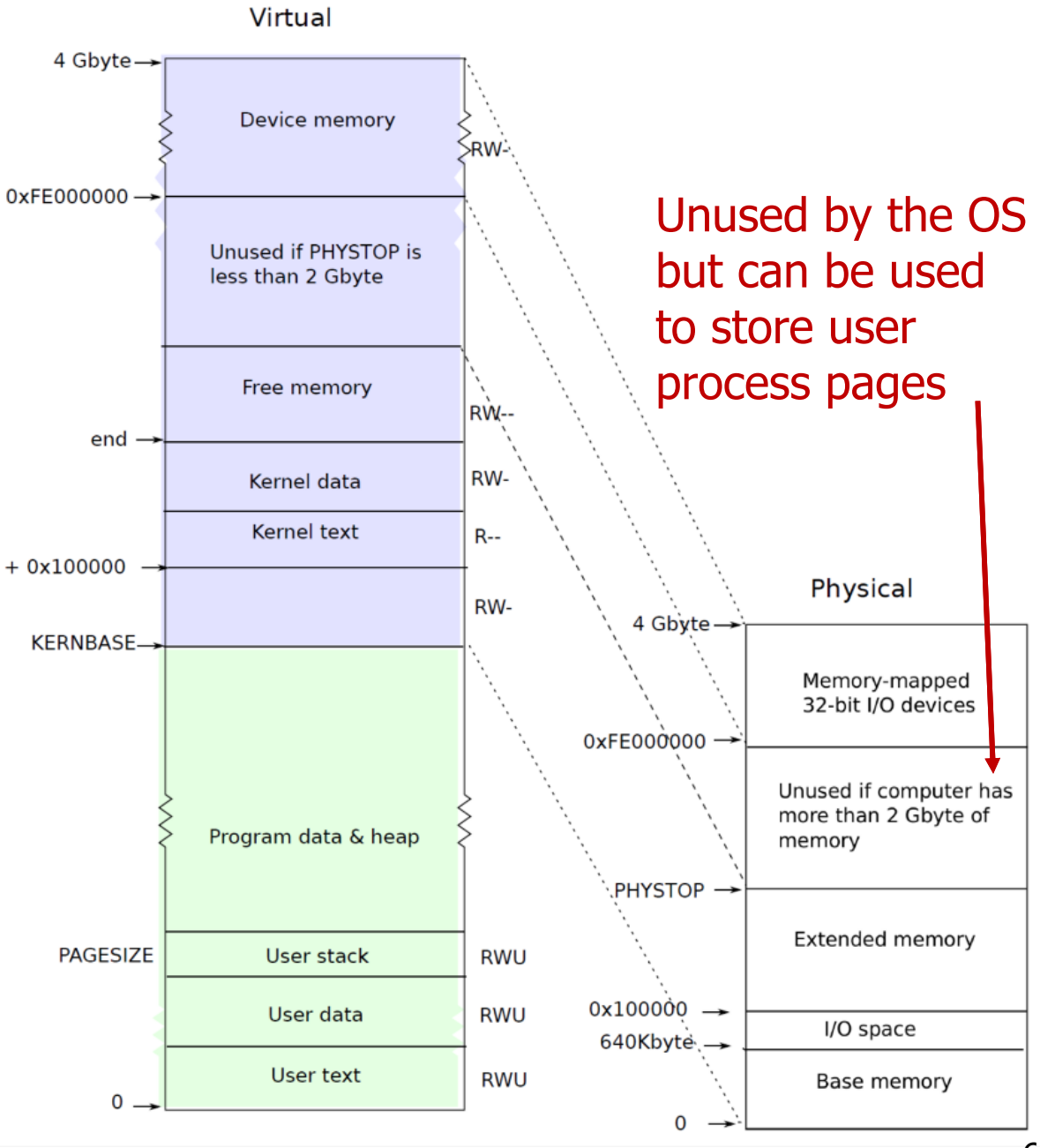
# xv6 Process Address Space (2 of 4)

- `memlayout.h` declares the constants for xv6's memory layout, and macros to convert between virtual and physical addresses.
- Every process's page table contains **mappings for both user memory and the entire kernel**. This is convenient since switching from user code to kernel code during system calls **won't require page table switching**.



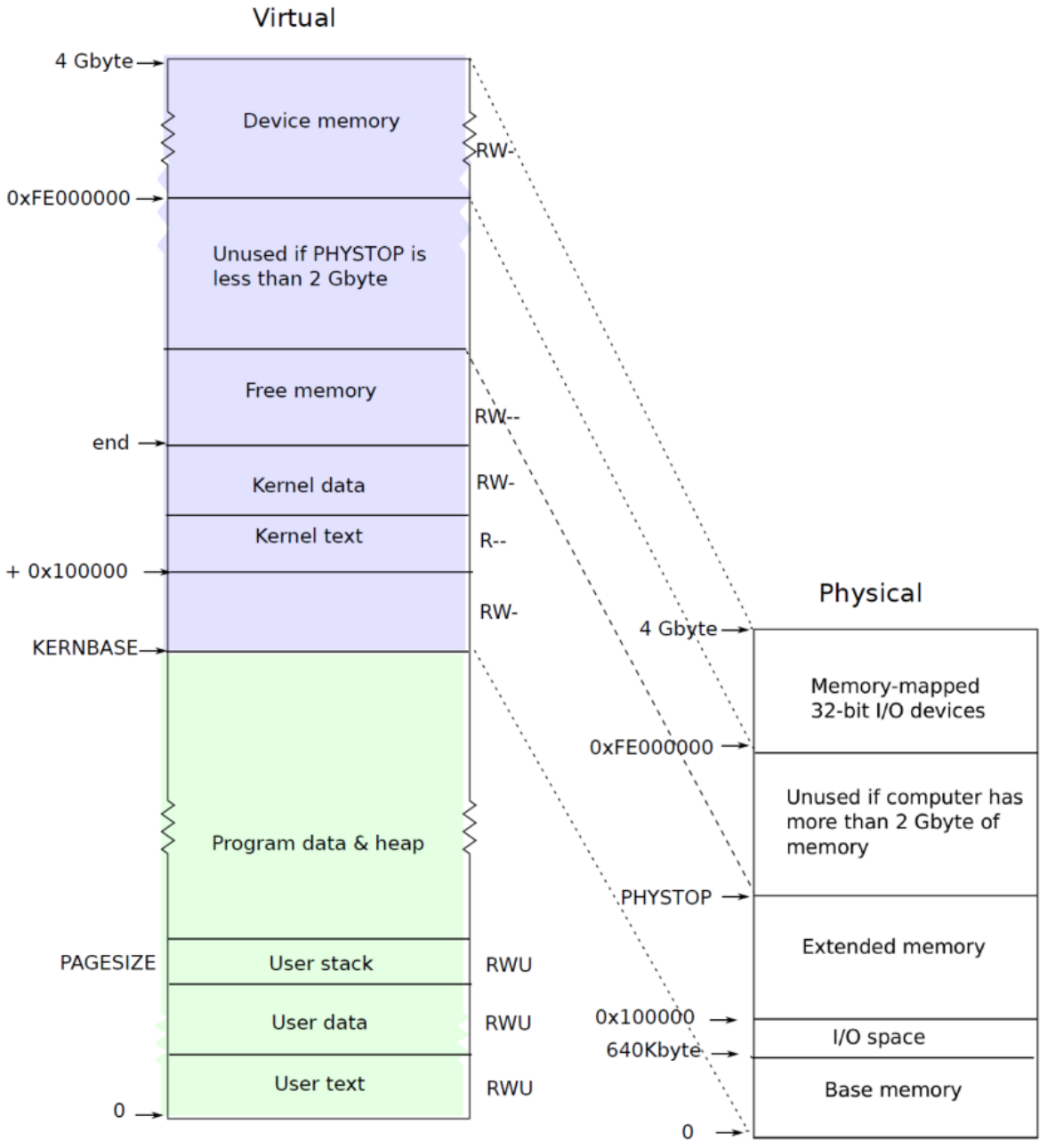
# xv6 Process Address Space (3 of 4)

- xv6 includes all mappings needed for the kernel to run in every process's page table.
- The part of the page table dealing with kernel pages is the same across all processes.
- `setupkvm()` and `mappages()` functions in "`vm.c`" map the kernel's physical memory addresses space (0 to PHYSTOP) into a new process's virtual addresses space (`KERNBASE` to `KERNBASE+PHYSTOP`) by setting up its page directory.



# xv6 Process Address Space (4 of 4)

- The layout of the user memory of an executing process in xv6 is shown in green.
- xv6 allocates only one `PAGESIZE` stack (i.e., 4 KB) for each process.
- It allocates heap above the stack, which can grow to `KERNBASE` (2 GB) when the process calls `sbrk` (explained later).



# Kernel Page Table

- There is one page table per process, plus one (`kpgdir`) that's used when a CPU is not running any process.
  - `kpgdir` is the kernel page table, it is used while running the `scheduler()`. The scheduler switches back to it by running `switchkvm()`, defined in `vm.c`
- As explained, the kernel uses the current process's page table during system calls and interrupts; page protection bits prevent user code from using the kernel's mappings.

```
switchuvm(p);  // switch to the process's page table (i.e., address space)
p->state = RUNNING;
swtch(&(cpu->scheduler), p->context);  //process p resumes running
switchkvm();  // switch back to the kernel page table
```

# xv6 Maintenance of Free Memory

- After boot up, RAM contains xv6 OS code/data and free pages.
- xv6 collects all free pages into a **free linked list**.
- The shown function `kalloc()` allocates the page at the head of this list, if available.
- Free pages are eventually assigned to code/data/stack/heap of user processes and page tables of these processes.
- Free list is a linked list where the kernel maintains pointer to first page in the list.

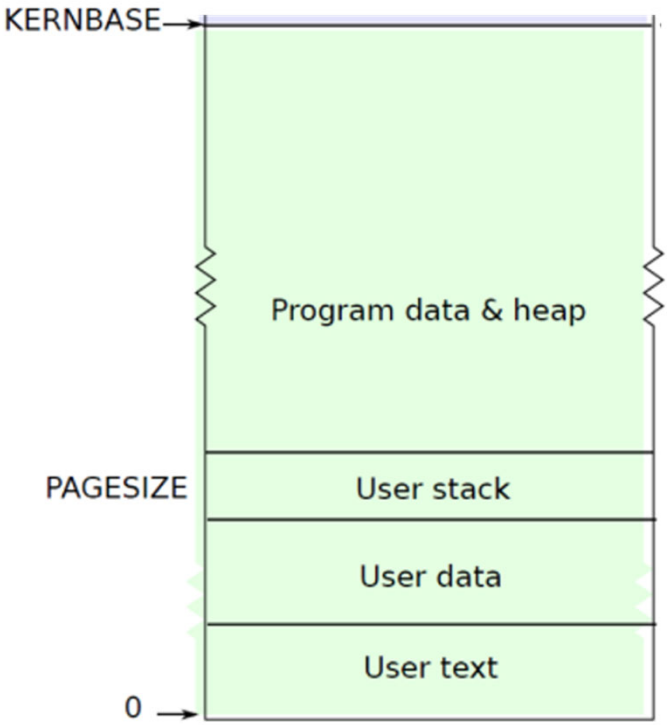
```
struct run {
  struct run *next;
};

struct {
  struct spinlock lock;
  int use_lock;
  struct run *freelist;
} kmem;

// Allocate one 4096-byte page of physical memory.
// Returns a pointer that the kernel can use.
// Returns 0 if the memory cannot be allocated.
char* kalloc(void) {
  struct run *r;
  if(kmem.use_lock)
    acquire(&kmem.lock);
  r = kmem.freelist;
  if(r) kmem.freelist = r->next;
  if(kmem.use_lock)
    release(&kmem.lock);
  return (char*)r;
}
```

# Code: exec

- `exec` is the system call (in `exec.c`) that creates the user part of an address space. It initializes this part from a file stored in the file system by first reading its ELF header.
  - **ELF** stands for ***Executable and Linkable Format*** and it is a standard binary file format and xv6 defines it in `elf.h`
- The first step is a quick check that the file probably contains an ELF binary, which must start with the four-byte "**magic number**" 0x7F, 'E', 'L', 'F'. If the ELF header has the right magic number, exec assumes that the binary is well-formed.



# Building a Page Table in exec

- The page table of a process is built in **exec** in three main steps as shown.
- We have already talked about step 1.

```
int
exec(char *path, char **argv)
{
  char *s, *last;
  int i, off;
  uint argc, sz, sp, ustack[3+MAXARG+1];
  struct elfhdr elf;
  struct inode *ip;
  struct proghdr ph;
  pde_t *pgdir, *oldpgdir;

  if((pgdir = setupkvm()) == 0)
    goto bad;

  // Load program into memory.
  sz = 0;
  for(i=0, off=elf.phoff; i<elf.phnum; i++, off+=sizeof(ph)){
    if(readi(ip, (char*)&ph, off, sizeof(ph)) != sizeof(ph))
      goto bad;
    if(ph.type != ELF_PROG_LOAD)
      continue;
    if(ph.memsz < ph.filesz)
      goto bad;
    if((sz = allocuvm(pgdir, sz, ph.vaddr + ph.memsz)) == 0)
      goto bad;
    if(loaduvm(pgdir, (char*)ph.vaddr, ip, ph.off, ph.filesz) < 0)
      goto bad;
  }

  // Allocate two pages at the next page boundary.
  // Make the first inaccessible.  Use the second as the user stack.
  sz = PGROUNDUP(sz);
  if((sz = allocuvm(pgdir, sz, sz + 2*PGSIZE)) == 0)
    goto bad;
  clearpteu(pgdir, (char*)(sz - 2*PGSIZE));
  sp = sz;
```

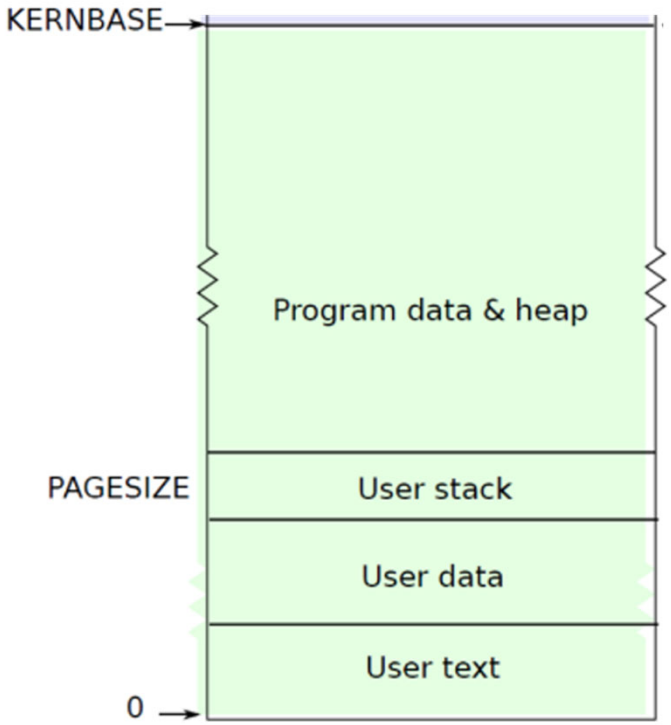
① map kernel to the page table

② Load the program image and map it to the page table

③ Initialize the runtime stack and heap and map them into the page table

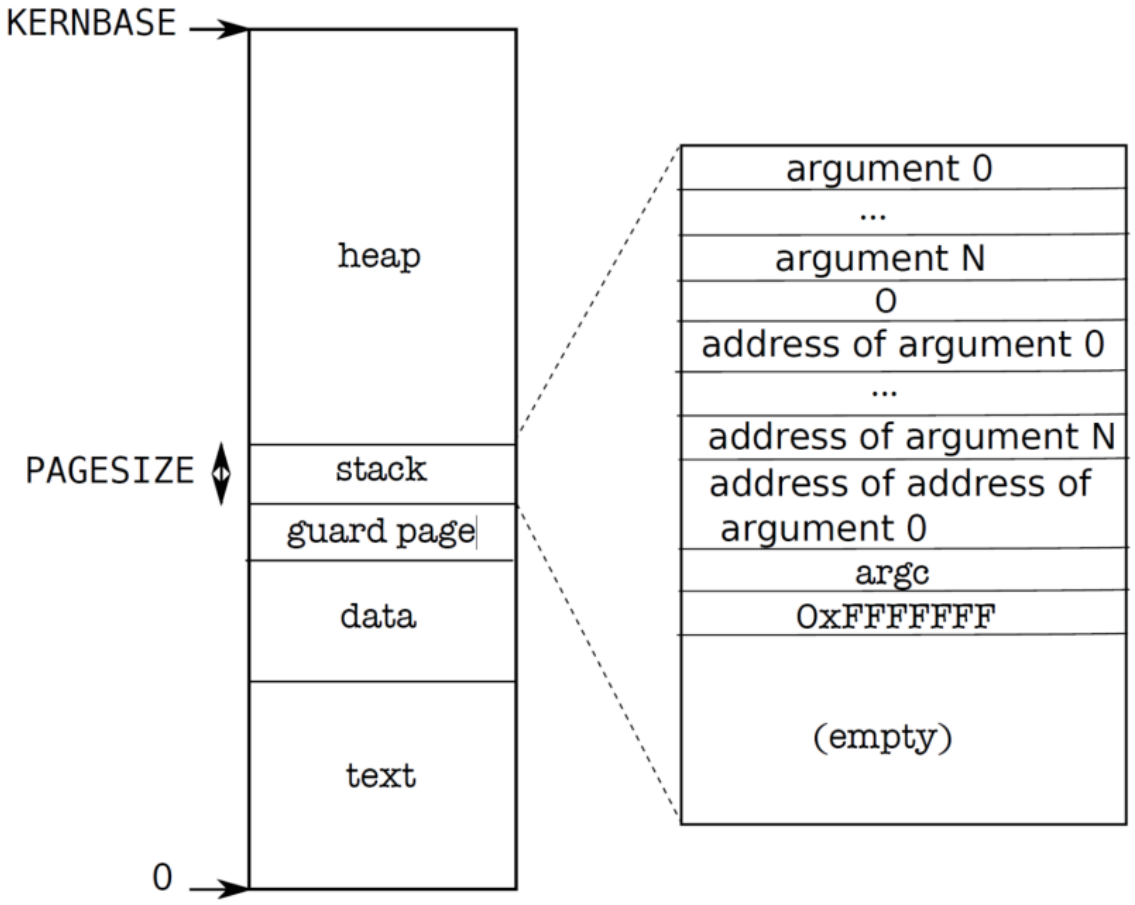
# Map User Program Text and Data

- **Step 2** in building a process page table is to map the user process data and text (i.e., its code) segments to the process page table.
- This is done by going over the program image (ELF file) segment by segment and loads segments marked as "`ELF_PROG_LOAD`".
- Function `allocuvm()` is used to allocate memory and map it to the page table.
- Function `loaduvm()` is used to load the segment to the allocated memory.



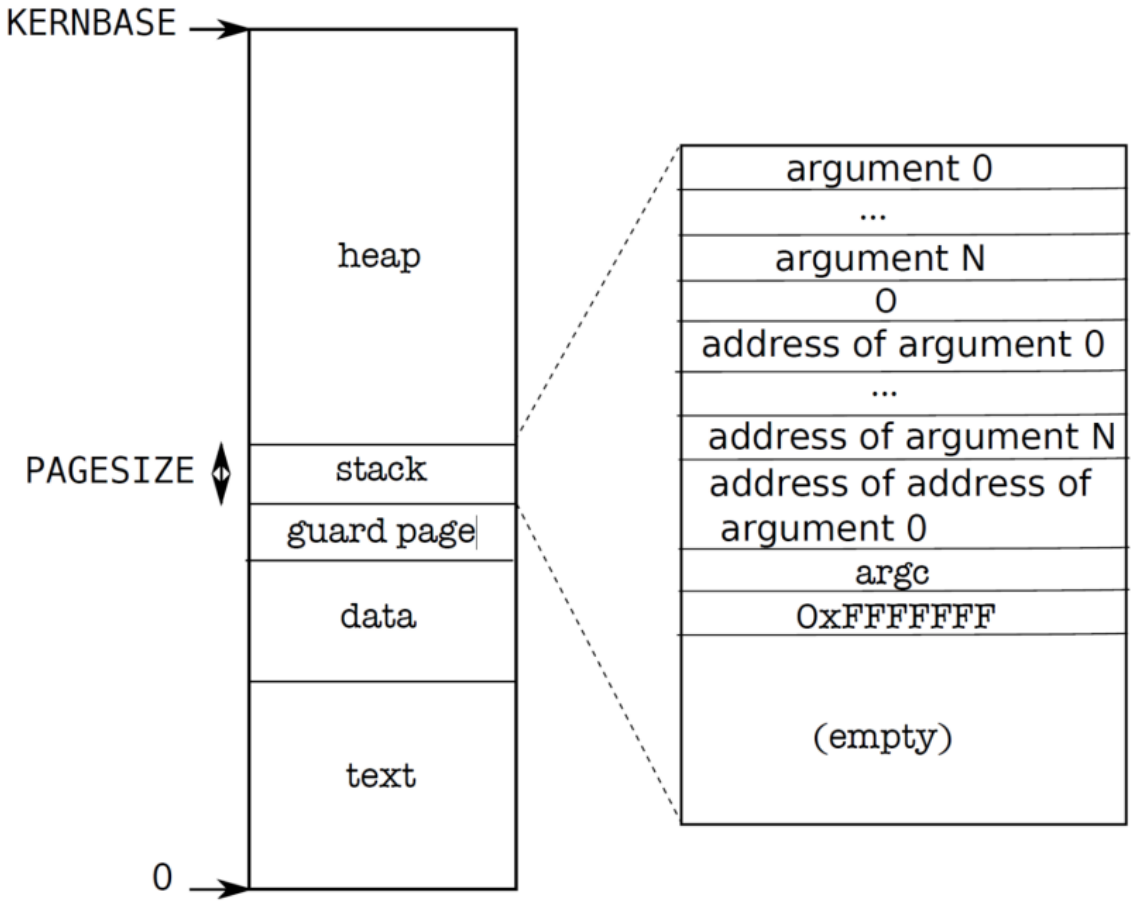
# Map User Stack and Heap (1 of 2)

- **Step 3** in building a process page table is to map the user stack and heap to the table.
- The stack is a single page, and is shown with the initial contents as created by `exec`.
- Strings containing the **command-line arguments**, an array of pointers to them, and `argc` are at the very top of the stack.



# Map User Stack and Heap (2 of 2)

- To guard a stack growing off the stack page, xv6 places a guard page right below the stack. The guard page is not mapped. If the stack runs off the stack page, the hardware will generate an **exception**.
- The **return address** "`0xffffffff`" is fake and is not allocated. Hence, in a xv6's program, using "**return**" in the main function will result in an error. So, use `exit()` instead.



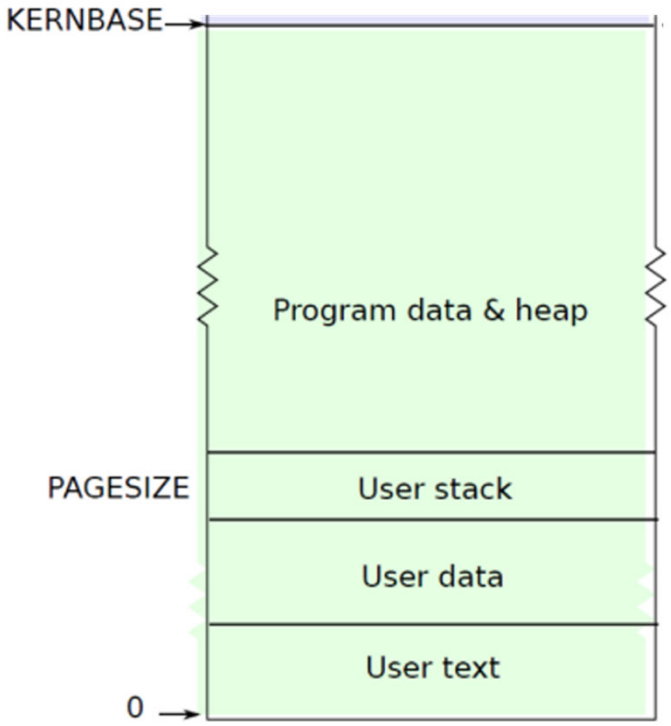
# Checking a Process Stack Address

- Add the shown two "`cprintf`" statements in `exec.c` to check the base address where the stack start and the maximum limit of the stack for any process that is executed by `exec`.
- Observe the output of these statements when you start xv6 as exec starts the "`init`" and "`shell`" processes.

```
56   // Allocate two pages at the next page boundary.
57   // Make the first inaccessible.  Use the second as the user stack.
58   sz = PGROUNDUP(sz);
59   if((sz = allocuvm(pgdir, sz, sz + 2*PGSIZE)) == 0)
60     goto bad;
61   clearpteu(pgdir, (char*)(sz - 2*PGSIZE));
62   sp = sz;
63   cprintf("pid:%d, base addr of the stack:0x%x\n", proc->pid, sp-1);
64   cprintf("pid:%d, end addr of the stack:0x%x\n", proc->pid, sp-PGSIZE);
65
66   // Push argument strings, prepare rest of stack in ustack.
67   for(argc = 0; argv[argc]; argc++) {
68     if(argc >= MAXARG)
69       goto bad;
70     sp = (sp - (strlen(argv[argc]) + 1)) & ~3;
71     if(copyout(pgdir, sp, argv[argc], strlen(argv[argc]) + 1) < 0)
72       goto bad;
73     ustack[3+argc] = sp;
74   }
75   ustack[3+argc] = 0;
```

# Code: `sbrk`

- **`sbrk`** is the system call for a process to shrink or grow its heap memory space.
- The system call is implemented by the function **`growproc(int n)`** in **`proc.c`**
  - If `n` is positive, one or more physical pages are allocated and mapped at the top of the process's address space.
  - If `n` is negative, **`growproc`** un-maps one or more pages from the process's address space and frees the corresponding physical pages



# Checking a Process Heap Size (1 of 2)

- Add the shown "**cprintf**" statement in **sysproc.c** to check how a process heap allocated size changes with calling malloc() from the process.
- Observe the output of these statements when you start xv6 as exec executes the shell.

```
int sys_sbrk(void) {
  int addr;
  int n;

  if(argint(0, &n) < 0) return -1;
  addr = myproc()->sz;
  if(growproc(n) < 0)
    return -1;
    cprintf("pid:%d grows heap from %x to
       %x with size of %d bytes.\n",
       proc -->pid,addr, proc -->sz, proc -->sz addr);
  return addr;
}
```

# Checking a Process Heap Size (2 of 2)

- To test how the heap size change, you need to add the following `test_heap.c` program as a new user command (the same way we added `hello.c` at the beginning of these slides).
- Run the command and observe the messages of the heap sizes.
- Replace the value 5 in malloc with 32768 and observe the difference.

```
#include "types.h"
#include "stat.h"
#include "user.h"
int main(int argc, char *argv[]) {
  int i = 10;
  while(i>0){
       char* addr = malloc(5 * sizeof(char));
       printf(1, "return addr:%p\n", addr);
       i--;
  }
  exit();
}
```


# Readings and Resources List

- xv6 Manual: https://pdos.csail.mit.edu/6.S081/2020/xv6/book-riscv-rev1.pdf
- xv6 Sources booklet: https://pdos.csail.mit.edu/6.828/2018/xv6/xv6-rev11.pdf
- xv6 Source Code Files: https://github.com/mit-pdos/xv6-public
- Intel® IA-32 Architectures Software Developer Manuals: https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html