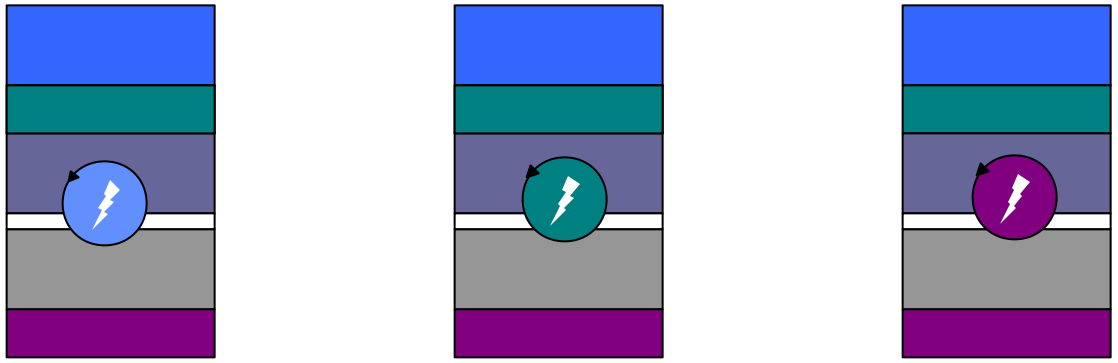
Processes, Protection and the Kernel:

Mode, Space, and Context

Processes and the Kernel

processes in private virtual address spaces

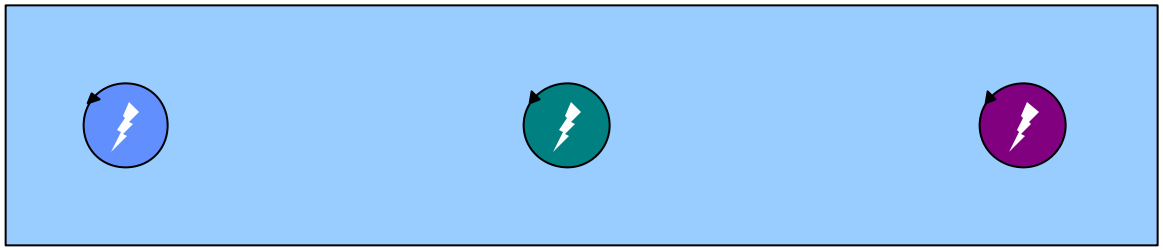


The kernel sets up process execution contexts to "virtualize" the machine.

system call traps

...and upcalls (e.g., signals)

shared kernel code and data in shared address space



Threads or processes enter the kernel for services.

CPU and devices force entry to the kernel to handle exceptional events.

Objectives

- The nature of the classical kernel, its protection mechanisms, and architectural support for protected kernels.
  - Mode, space, and context.
- Control transfer from user code into the kernel.
  - System calls (traps) and user program events (faults).
  - Access control: handles, IDs, and Access Control Lists.
- Control transfer from the kernel to user code.
  - Signals, APCs, syscall return.
- Kernel synchronization.
- Process structure and process birth/death, process states.
  - Fork/exec/exit/join/wait and process trees.

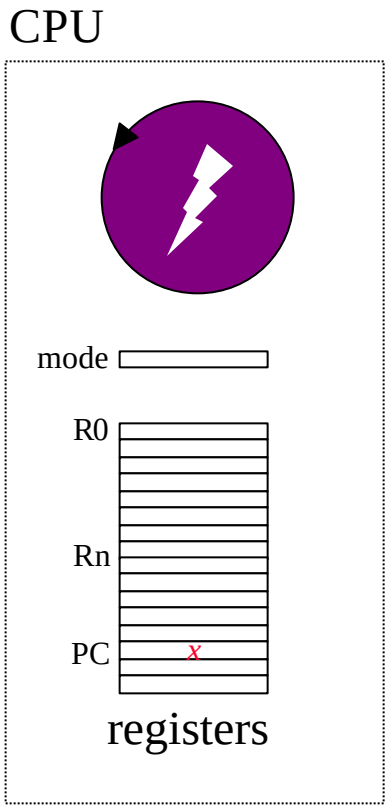
The Kernel

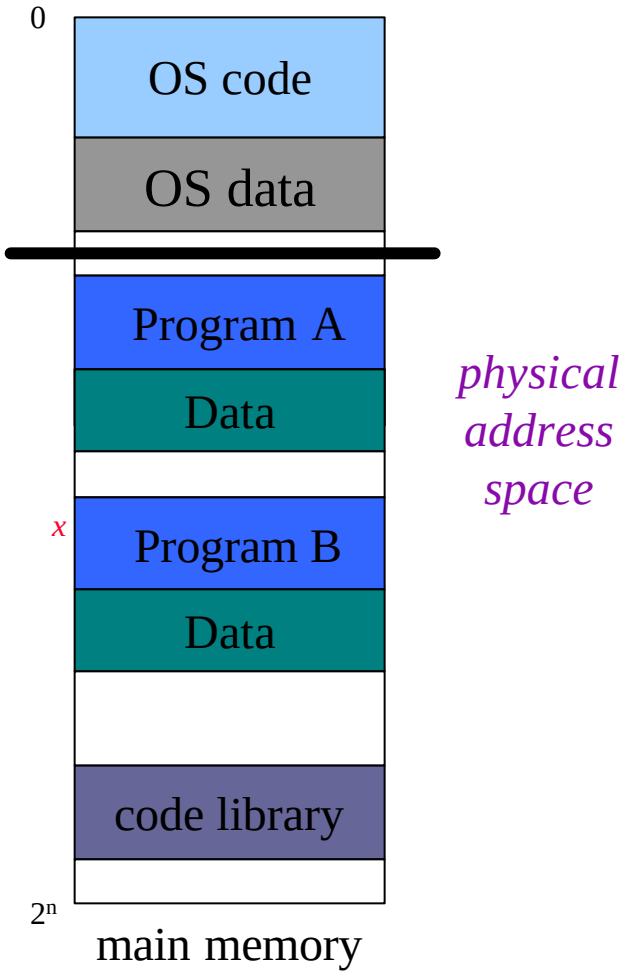
- Today, all “real” operating systems have protected kernels.
  - The kernel resides in a well-known file: the “machine” automatically loads it into memory (*boots*) on power-on/reset.
  - Our “kernel” is called the *executive* in some systems (e.g., MS).
- The kernel is (mostly) a library of service procedures shared by all user programs, *but the kernel is* ***protected***:
  - User code cannot access internal kernel data structures directly, and it can invoke the the kernel only at well-defined entry points (*system calls*).
- Kernel code is like user code, but the kernel is ***privileged***:
  - The kernel has direct access to all hardware functions, and defines the entry points of handlers for *interrupts* and *exceptions (traps* and *faults)*.

Kernel Mode

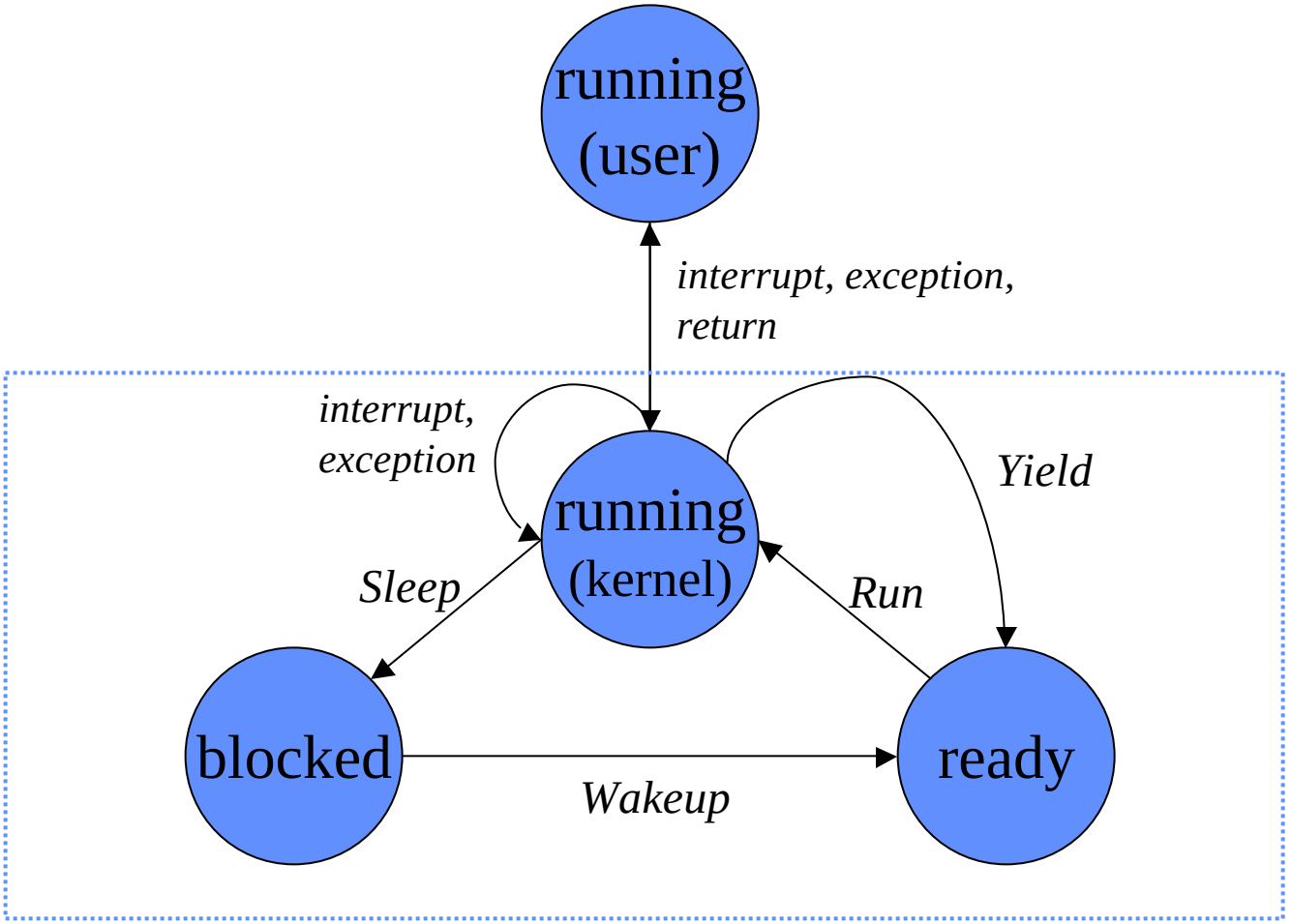
CPU *mode* (a field in some status register) indicates whether the CPU is running in a *user* program or in the protected *kernel*.

Some instructions or data accesses are only legal when the CPU is executing in kernel mode.





Thread/Process States and Transitions



CPU Events: Interrupts and Exceptions

An *interrupt* is caused by an external event.

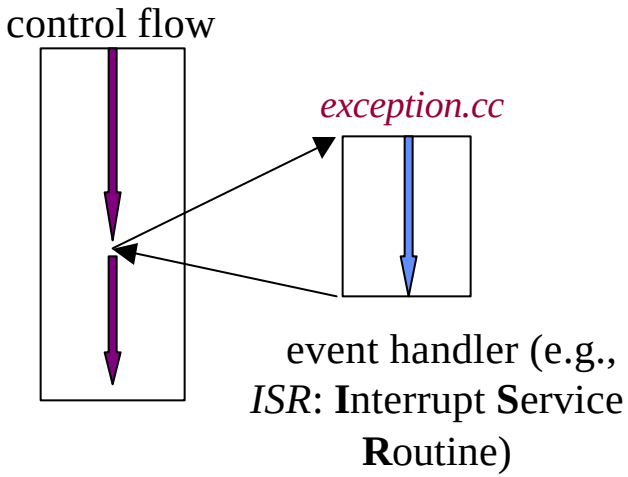
device requests attention, timer expires, etc.

An *exception* is caused by an executing instruction.

CPU requires software intervention to handle a *fault* or *trap*.

	unplanned	deliberate
sync	*fault*	*syscall trap*
async	*interrupt*	AST

AST: **A**synchronous **S**ystem **T**rap
Also called a *software interrupt* or an Asynchronous or Deferred Procedure Call (APC or DPC)



Note: different "cultures" may use some of these terms (e.g., trap, fault, exception, event, interrupt) slightly differently**.**

Protecting Entry to the Kernel

Protected events *and* ***kernel mode*** *are the architectural foundations of kernel-based OS (Unix, NT+, etc).*

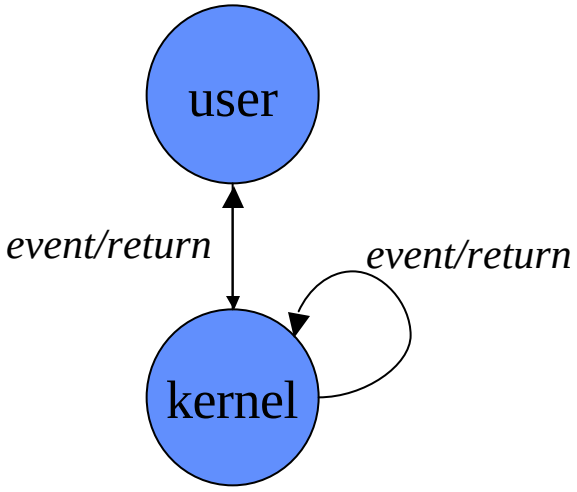
- The *machine* defines a small set of exceptional event types.
- The *machine* defines what conditions raise each event.
- The kernel installs handlers for each event at boot time.

  e.g., a table in kernel memory read by the machine

The machine transitions to kernel mode only on an exceptional event.

The kernel defines the event handlers.

Therefore the *kernel* chooses what code will execute in kernel mode, and when.



Handling Events, Part I: The Big Picture

1. To deliver the event, the machine saves relevant state in temporary storage, then transfers control to the kernel.

   Set kernel mode and set PC := *handler*.

2. Kernel handler examines registers and saved machine state.

   What happened? What was the machine doing when it happened? How should the kernel respond?

3. Kernel responds to the condition.

   Execute kernel service, device control code, fault handlers, etc., modify machine state as needed.

4. Kernel restores saved context (registers) and resumes activity.

5. Specific events and mechanisms for saving, examining, or restoring context are *machine-dependent*.

The Role of Events

Once the system is booted, *every entry to the kernel occurs as a result of an event.*

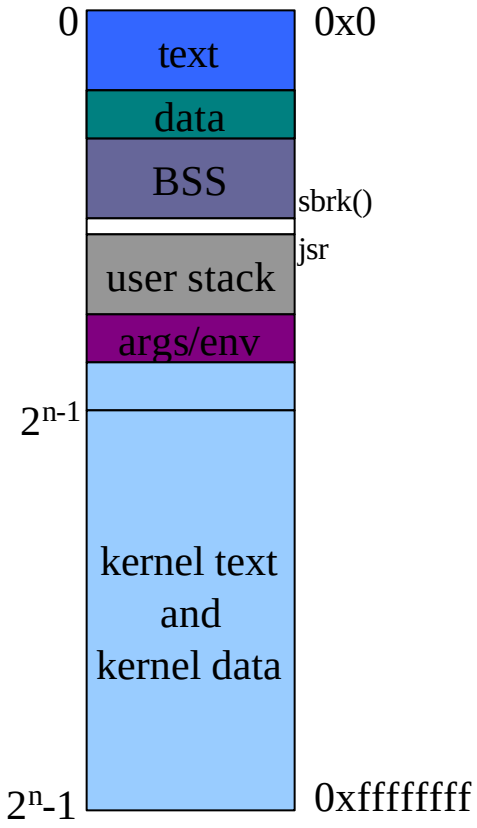
- In some sense, the whole kernel is a big event handler.
- Event handlers are kernel-defined and execute in kernel mode.
- Events do *not* change the identity of the executing thread/process.

  Context: thread/process context, or interrupt context.

  Loosely, whose stack are you running on.

  For purposes of this discussion, suppose one thread per process.
- Events do *not* change the current space!

The Virtual Address Space

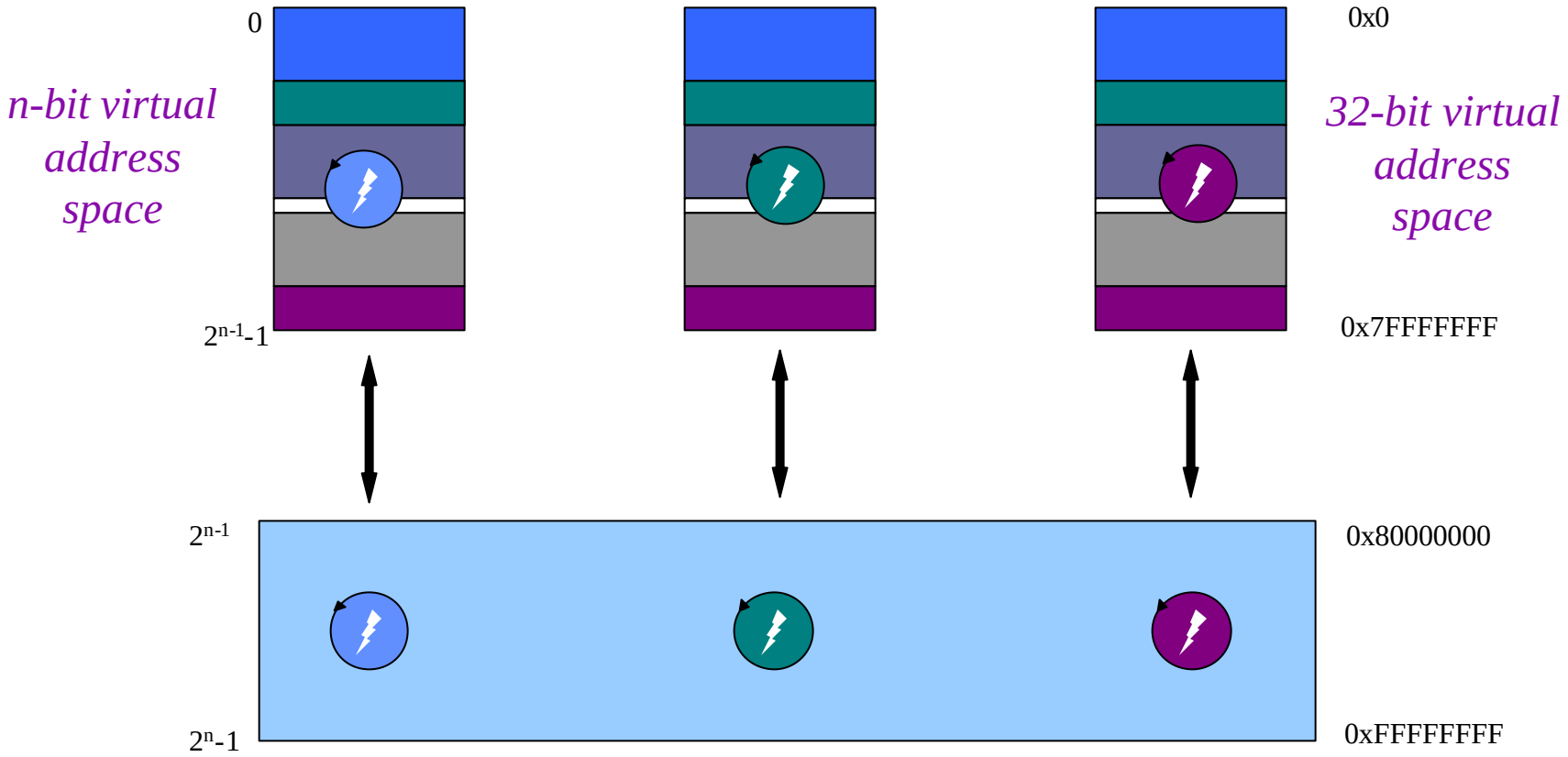


A *typical* process VAS space includes:

- user regions in the lower half
  - V->P mappings specific to each process
  - accessible to user or kernel code
- kernel regions in upper half
  - shared by all processes
  - accessible only to kernel code
- **Nachos:** process virtual address space includes only user portions.

A VAS for a private address space system (e.g., Unix) executing on a typical 32-bit architecture.

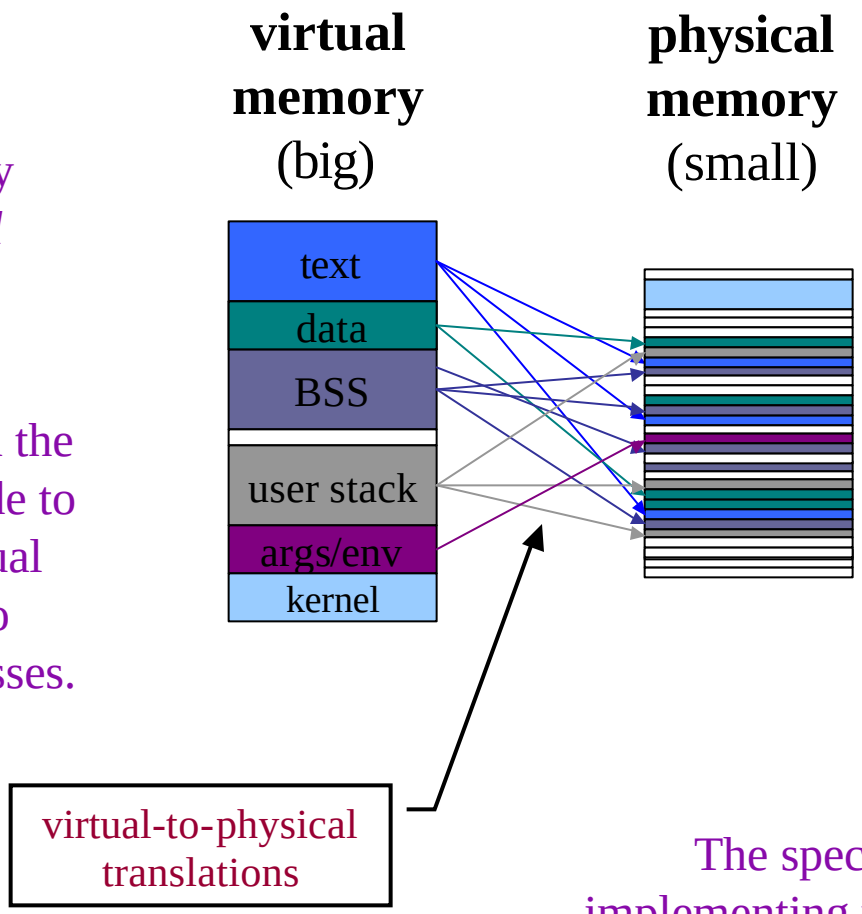
Example: Process and Kernel Address Spaces



Introduction to Virtual Addressing

User processes address memory through *virtual addresses.*

The kernel and the machine collude to translate virtual addresses to physical addresses.



The kernel controls the virtual-physical translations in effect for each space.

The machine does not allow a user process to access memory unless the kernel "says it's OK".

The specific mechanisms for implementing virtual address translation are *machine-dependent*: we will cover them later.

System Call Traps

User code invokes kernel services by initiating *system call* traps.

- Programs in C, C++, etc. invoke system calls by linking to a *standard library* of procedures written in assembly language.

  The library defines a *stub* or *wrapper* routine for each syscall.

  Stub executes a special **trap** instruction (e.g., **chmk** or **callsys**).

  Syscall arguments/results passed in registers or user stack.

read() in Unix libc.a library (executes in user mode): Alpha CPU architecture

```
#define SYSCALL_READ 27          # number for a read system call
move arg0…argn, a0…an    # syscall args in registers A0..AN
move SYSCALL_READ, v0    # syscall dispatch code in V0
callsys                  # kernel trap
move r1, _errno          # errno = return status
return
```

“Bullet-Proofing” the Kernel

System calls must be “safe” to protect the kernel from buggy or malicious user programs.

1. System calls enter the kernel at a well-known safe point.

   Enter at the kernel trap handler; control transfers to the “middle” of the kernel are not permitted.

2. The kernel validates all system call arguments before use.

   Kernel may reject a request if it is meaningless or if the user process has inadequate privilege for the requested operation.

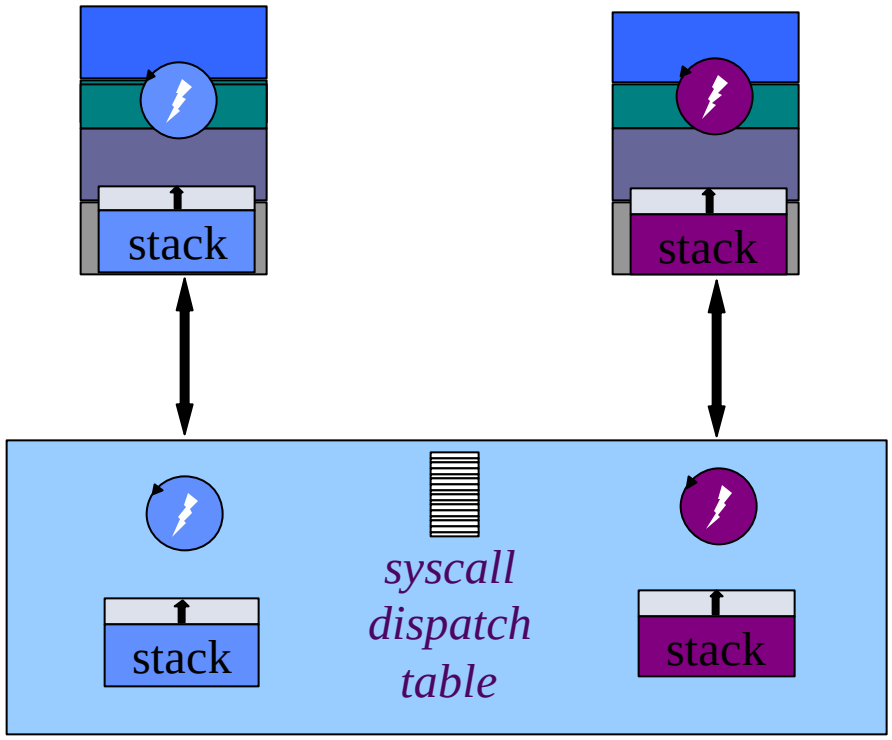
3. All memory used by the system call handler is in kernel space, so it is protected from interference by user code.

   What stack does the system call execute on?

Kernel Stacks and Trap/Fault Handling

Processes execute user code on a ***user stack*** in the user portion of the process virtual address space.

Each process has a second ***kernel stack*** in kernel space (the kernel portion of the address space).



System calls and faults run in kernel mode on the process kernel stack.

System calls run in the process space, so *copyin* and *copyout* can access user memory.

The syscall trap handler makes an indirect call through the *system call dispatch* table to the handler for the specific system call.

Safe Handling of Syscall Args/Results

1. Decode and validate by-value arguments.

   Process (stub) leaves arguments in registers or on the stack.

2. Validate by-reference (pointer) IN arguments.

   Validate user pointers and copy data into kernel memory with a special safe copy routine, e.g., *copyin()*.

3. Validate by-reference (pointer) OUT arguments.

   Copy OUT results into user memory with special safe copy routine, e.g., *copyout()*.

4. Set up registers with return value(s); return to user space.

   Stub may check to see if syscall failed, possibly raising a user program exception or storing the result in a variable.

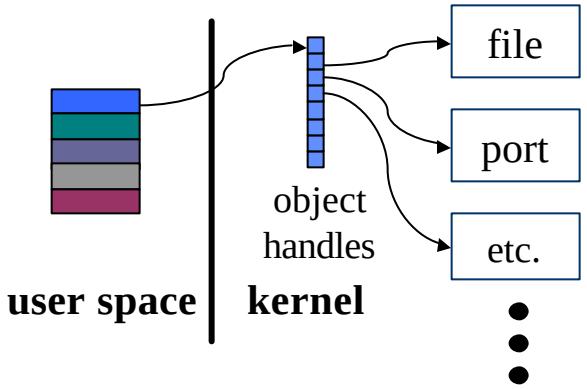
Kernel Object Handles

Instances of kernel abstractions may be viewed as "objects" named by protected *handles* held by processes.

- Handles are obtained by *create/open* calls, subject to security policies that grant specific rights for each handle.
- Any process with a handle for an object may operate on the object using operations (system calls).

  Specific operations are defined by the object's type.
- The handle is an integer index to a kernel table.

Microsoft NT object handles
Unix file descriptors
Nachos *FileID* and *SpaceID*



Example: Mechanics of an Alpha Syscall Trap

1. *Machine* saves return address and switches to kernel stack.
   - save user SP, global pointer(GP), PC on kernel stack
   - *set kernel mode* and transfer to a syscall trap handler (*entSys*)
2. *Trap handler* saves software state, and dispatches.
   - save some/all registers/arguments on process kernel stack
   - vector to syscall routine through *sysent[v0: dispatchcode]*
3. Trap handler returns to user mode.
   - when syscall routine returns, restore user register state
   - execute privileged return-from-syscall instruction (*retsys*)
   - machine restores SP, GP, PC and sets user mode
   - emerges at user instruction following the *callsys*

Questions About System Call Handling

1. Why do we need special *copyin* and *copyout* routines?

   validate user addresses before using them

2. What would happen if the kernel did not save all registers?

3. Where should per-process kernel global variables reside?

   syscall arguments (consider size) and error code

4. What if the *kernel* executes a **callsys** instruction? What if user code executes a **retsys** instruction?

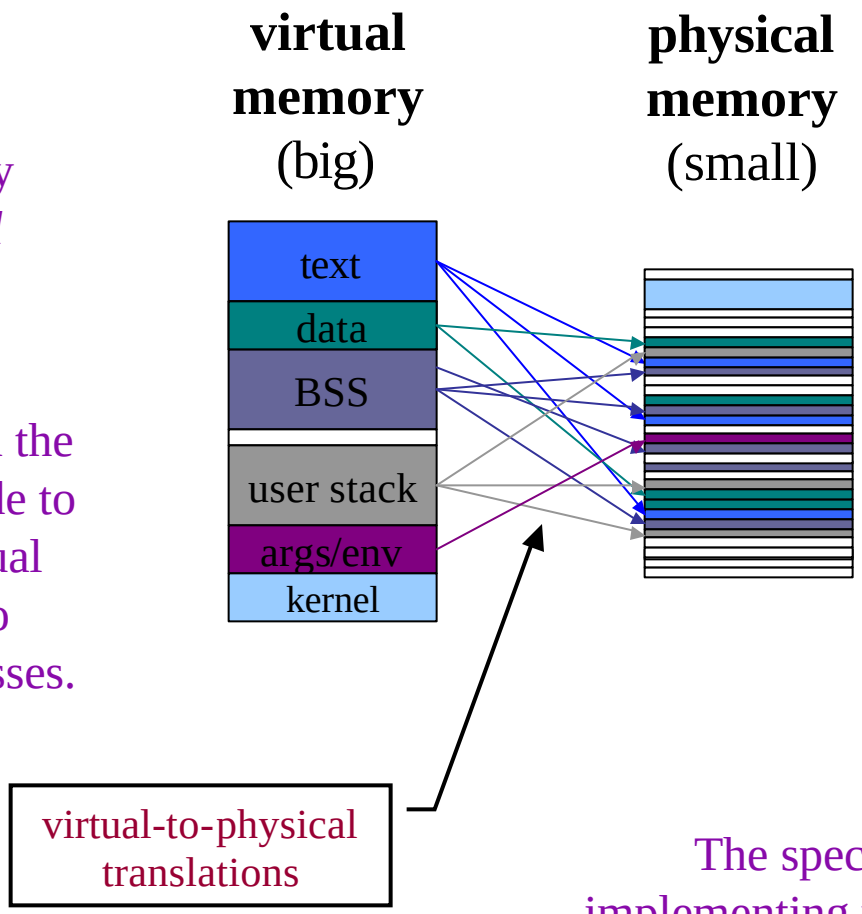
5. How to pass references to kernel objects as arguments or results to/from system calls?

   pointers? **No**: use integer *object handles* or *descriptors* (also sometimes called *capabilities).*

Flashback: Virtual Addressing

User processes address memory through *virtual addresses.*

The kernel and the machine collude to translate virtual addresses to physical addresses.

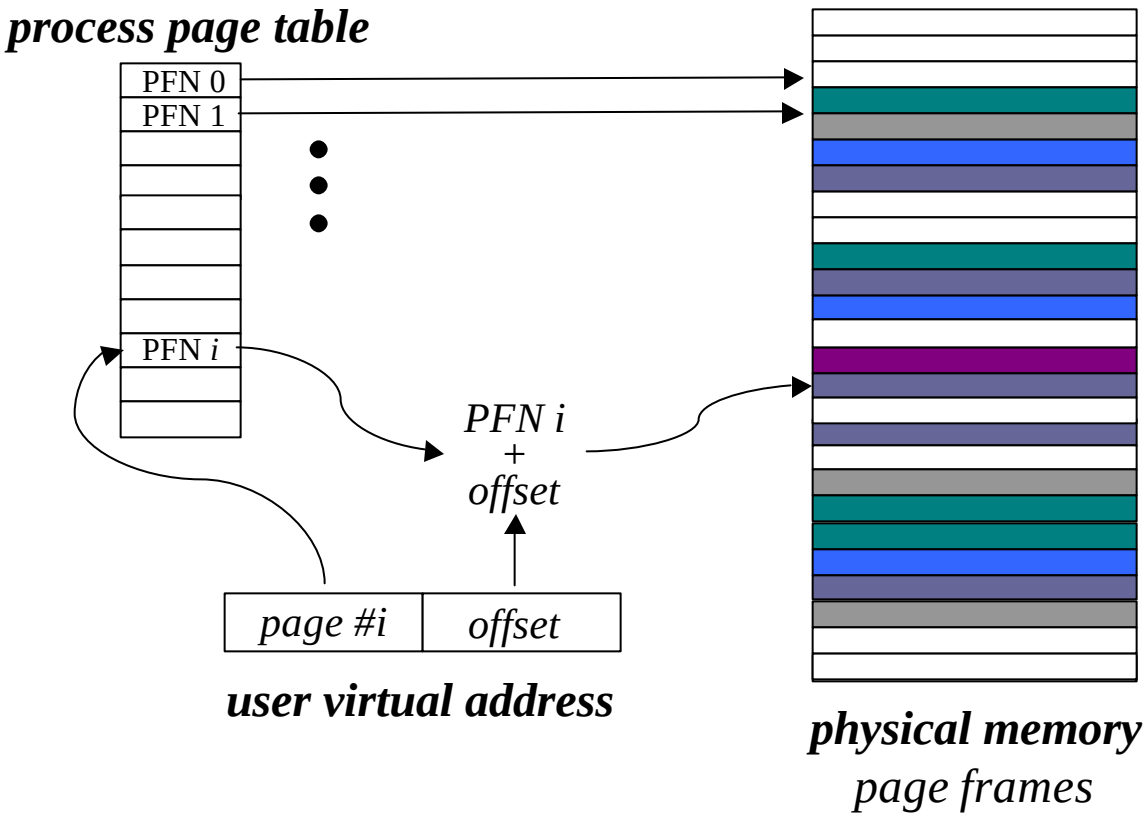


The kernel controls the virtual-physical translations in effect for each space.

The machine does not allow a user process to access memory unless the kernel "says it's OK".

The specific mechanisms for implementing virtual address translation are *machine-dependent*: we will cover them later.

A Simple Page Table



Each process/VAS has its own page table. Virtual addresses are translated relative to the current page table.

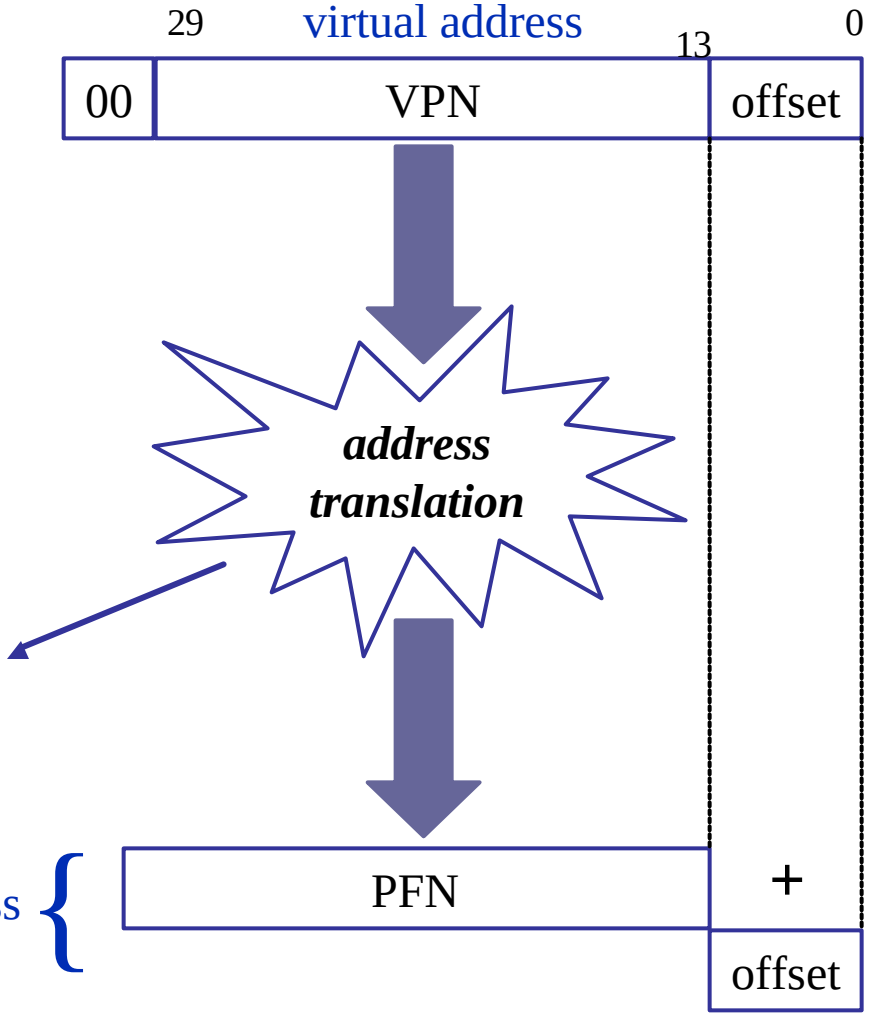
In this example, each VPN *j* maps to PFN *j*, but in practice any physical frame may be used for any virtual page.

The page tables are themselves stored in memory; a protected register holds a pointer to the current page table.

Virtual Address Translation

Example: typical 32-bit architecture with 8KB pages.

Virtual address translation maps a *virtual page number* (VPN) to a physical *page frame number* (PFN): the rest is easy.



Faults

Faults are similar to system calls in some respects:

- Faults occur as a result of a process executing an instruction.

  Fault handlers execute on the process kernel stack; the fault handler may block (sleep) in the kernel.

- The completed fault handler may return to the faulted context.

But faults are different from syscall traps in other respects:

- Syscalls are deliberate, but faults are “accidents”.

  divide-by-zero, dereference invalid pointer, memory page fault

- Not every execution of the faulting instruction results in a fault.

  may depend on memory state or register contents

Options for Handling a Fault (1)

1. Some faults are handled by "patching things up" and returning to the faulted context.

   Example: the kernel may resolve an address fault (virtual memory fault) by installing a new virtual-physical translation.

   The fault handler may adjust the saved PC to re-execute the faulting instruction after returning from the fault.

2. Some faults are handled by notifying the process that the fault occurred, so it may recover in its own way.

   Fault handler munges the saved user context (PC, SP) to transfer control to a registered user-mode handler on return from the fault.

   Example: Unix *signals* or Microsoft NT *user-mode **A**synchronous **P**rocedure **C**alls (APCs).*

Options for Handling a Fault (2)

3. The kernel may handle unrecoverable faults by killing the user process.

   Program fault with no registered user-mode handler?

   Destroy the process, release its resources, maybe write the memory image to a file, and find another ready process/thread to run.

   In Unix this is the default action for many signals (e.g., SEGV).

4. How to handle faults generated by the kernel itself?

   Kernel follows a bogus pointer? Divides by zero? Executes an instruction that is undefined or reserved to user mode?

   These are generally fatal operating system errors resulting in a system crash, e.g., *panic()*!

Thought Questions About Faults

1. How do you suppose *ASSERT* and *panic* are implemented?
2. Unix systems allow you to run a program "under a debugger". How do you suppose that works?

   If the program crashes, the debugger regains control and allows you to examine/modify its memory and register values!

3. Some operating systems allow *remote debugging*. A remote machine may examine/modify a crashed system over the network. How?
4. How can a user-mode fault handler recover from a fault? How does it return to the faulted context?
5. How can a debugger restart a program that has stopped, e.g., due to a fault? How are breakpoints implemented?
6. What stack do signal handlers run on?

Architectural Foundations of OS Kernels

- One or more privileged execution modes (e.g., *kernel mode*)
  - protected device control registers
  - privileged instructions to control basic machine functions
- System call *trap* instruction and protected fault handling
  - User processes safely enter the kernel to access shared OS services.
- Virtual memory mapping
  - OS controls virtual-physical translations for each address space.
- Device interrupts to notify the kernel of I/O completion etc.
  - Includes timer hardware and clock interrupts to periodically return control to the kernel as user code executes.
- Atomic instructions for coordination on multiprocessors

A Few More Points about Events

The *machine* may actually be implemented by a combination of hardware and special pre-installed software (firmware).

- PAL (Privileged Architecture Library) on Alpha
  - hides hardware details from even the OS kernel
  - some instructions are really short PAL routines
  - some special "machine registers" are really in PAL scratch memory, not CPU registers

Events illustrate hardware/software tradeoffs:

- how much of the context should be saved on an event or switch, and by whom (hardware, PAL, or OS)
- *goal*: simple hardware and good performance in common cases

Mode, Space, and Context

At any time, the state of each processor is defined by:

1. *mode*: given by the mode bit

   Is the CPU executing in the protected kernel or a user program?

2. *space*: defined by V->P translations currently in effect

   What address space is the CPU running in? Once the system is booted, it always runs in some virtual address space.

3. *context*: given by register state and execution stream

   Is the CPU executing a thread/process, or an interrupt handler?

   Where is the stack?

These are important because the mode/space/context determines the meaning and validity of key operations.

Common Mode/Space/Context Combinations

1. *User code* executes in a process/thread context in a process address space, in user mode.

    Can address only user code/data defined for the process, with no access to privileged instructions.

2. *System services* execute in a process/thread context in a process address space, in kernel mode.

    Can address kernel memory or user process code/data, with access to protected operations: may sleep in the kernel.

3. *Interrupts* execute in a system interrupt context in the address space of the interrupted process, in kernel mode.

    Can access kernel memory and use protected operations.

    no sleeping!

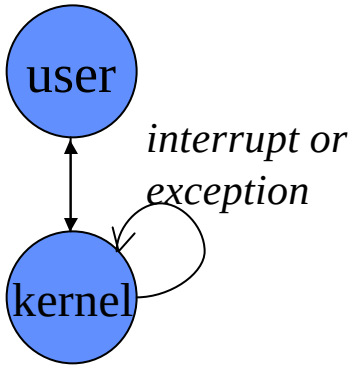
Kernel Concurrency Control 101

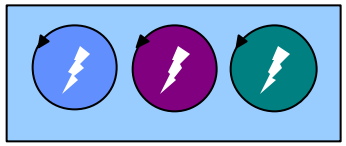
Processes/threads running in kernel mode share access to system data structures in the kernel address space.

- *Sleep/wakeup* (or equivalent) are the basis for:

  coordination, e.g., join (*exit/wait*), timed waits (*pause*), bounded buffer (pipe *read/write*), message *send/receive*

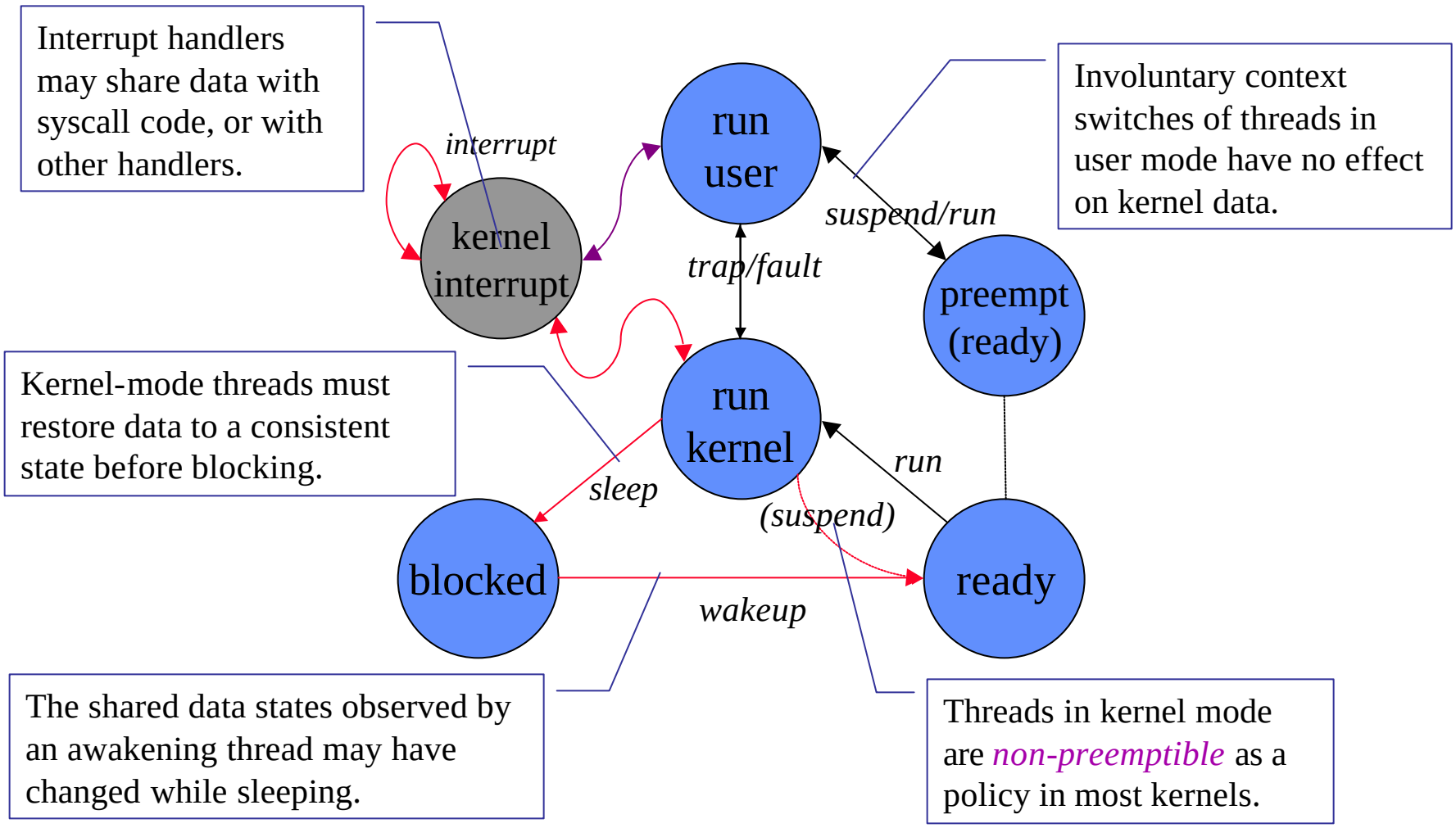
  synchronization, e.g., long-term mutual exclusion for atomic *read*/write** syscalls





Sleep/wakeup is sufficient for concurrency control among kernel-mode threads on uniprocessors: problems arise from *interrupts* and *multiprocessors.*

Dangerous Transitions



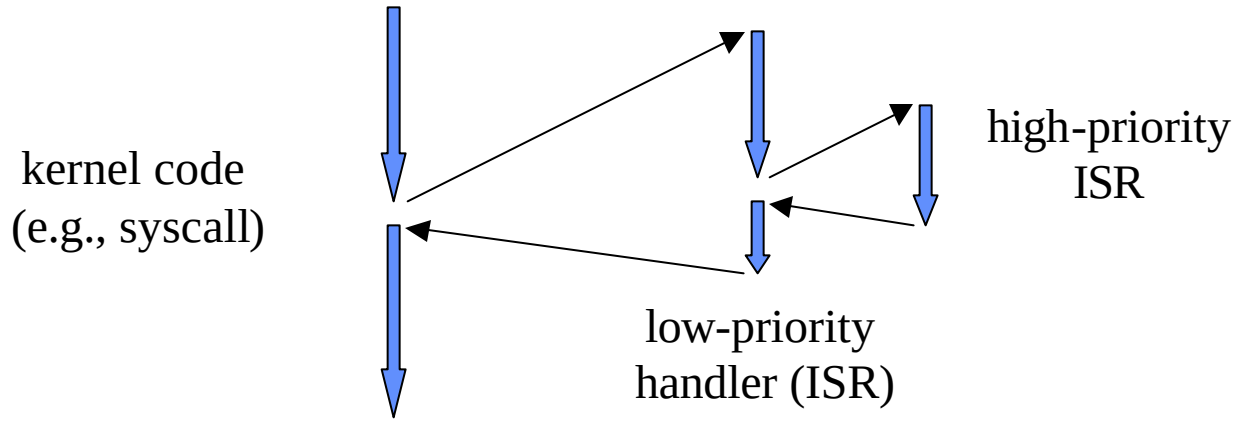
The Problem of Interrupts

Interrupts can cause races if the handler (ISR) shares data with the interrupted code.

e.g., *wakeup* call from an ISR may corrupt the sleep queue.

Interrupts may be nested.

ISRs may race with each other.



Interrupt Priority

Traditional Unix kernels illustrate the basic approach to avoiding interrupt races.

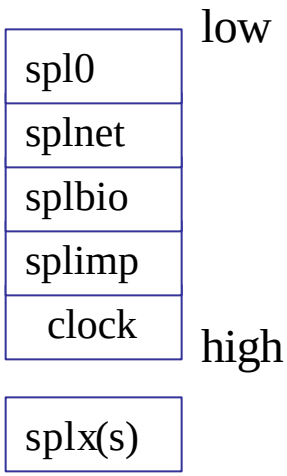
- Rank interrupt types in *N priority classes*.
- When an ISR at priority *p* runs, CPU blocks interrupts of priority *p* or lower.

  How big must the interrupt stack be?

- Kernel software can query/raise/lower the CPU *interrupt priority level* (IPL).

  Avoid races with an ISR of higher priority by raising CPU IPL to that priority.

  Unix *spl*/splx* primitives (may need software support on some architectures).

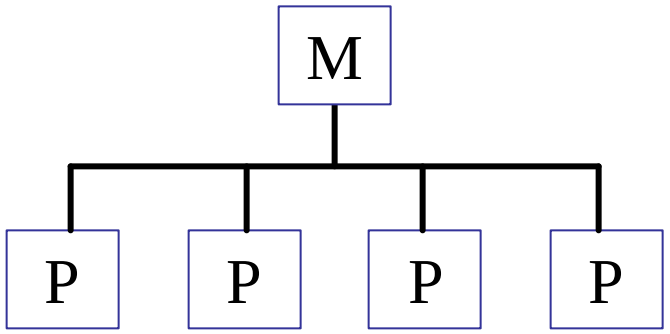



```
int s;
s = splhigh();
/* touch sleep queues */
splx(s);
```

Multiprocessor Kernels

On a shared memory multiprocessor, non-preemptible kernel code and *spl*()* are no longer sufficient to prevent races.

- **Option 1**, *asymmetric multiprocessing*: limit all handling of traps and interrupts to a single processor.
  - slow and boring
- **Option 2**, *symmetric multiprocessing* ("SMP"): supplement existing synchronization primitives.
  - any CPU may execute kernel code
  - synchronize with spin-waiting
  - requires atomic instructions
  - use *spinlocks…*
  - *…but still must disable interrupts*



Example: Unix Sleep

Optional

```
sleep (void* event, int sleep_priority)
{
        struct proc *p = curproc;
        int s;

        s = splhigh();                          /* disable all interrupts */
        p->p_wchan = event;                     /* what are we waiting for */
        p->p_priority -> priority;              /* wakeup scheduler priority */
        p->p_stat = SSLEEP;                     /* transition curproc to sleep state */
        INSERTQ(&slpque[HASH(event)], p);       /* fiddle sleep queue */
        splx(s);                                /* enable interrupts */
        mi_switch();                            /* context switch */
        /* we're back... */
}
```

Implementing Sleep on a Multiprocessor

Optional

```
sleep (void* event, int sleep_priority)
{
        struct proc *p = curproc;
        int s;

        s = splhigh();                          /* disable all interrupts */
        p->p_wchan = event;                     /* what are we waiting for */
        p->p_priority -> priority;              /* wakeup scheduler priority */
        p->p_stat = SSLEEP;                     /* transition curproc to sleep state */
        INSERTQ(&slpque[HASH(event)], p);       /* fiddle sleep queue */
        splx(s);                                /* enable interrupts */
        mi_switch();                            /* context switch */
        /* we're back... */
}
```

What if another CPU takes an interrupt and calls *wakeup*?

What if another CPU is handling a syscall and calls *sleep* or *wakeup*?

What if another CPU tries to *wakeup curproc* before it has completed *mi_switch*?

Using Spinlocks in *Sleep*: First Try

Optional

```
sleep (void* event, int sleep_priority)
{
        struct proc *p = curproc;
        int s;

        lock spinlock;
        p->p_wchan = event;                 /* what are we waiting for */
        p->p_priority -> priority;          /* wakeup scheduler priority */
        p->p_stat = SSLEEP;                 /* transition curproc to sleep state */
        INSERTQ(&slpque[HASH(event)], p);       /* fiddle sleep queue */
        unlock spinlock;
        mi_switch();                        /* context switch */
        /* we're back */
}
```

Grab spinlock to prevent another CPU from racing with us.

Wakeup (or any other related critical section code) will use the same spinlock, guaranteeing mutual exclusion.

Sleep with Spinlocks: What Went Wrong

Optional

```
sleep (void* event, int sleep_priority)
{
        struct proc *p = curproc;
        int s;

        lock spinlock;
        p->p_wchan = event;                 /* what are we waiting for */
        p->p_priority -> priority;          /* wakeup scheduler priority */
        p->p_stat = SSLEEP;                 /* transition curproc to sleep state */
        INSERTQ(&slpque[HASH(event)], p);   /* fiddle sleep queue */
        unlock spinlock;
        mi_switch();                        /* context switch */
        /* we're back */
}
```

Potential *deadlock*: what if we take an interrupt on this processor, and call *wakeup* while the lock is held?

Potential doubly scheduled thread: what if another CPU calls *wakeup* to wake us up before we're finished with *mi_switch* on this CPU?

Using Spinlocks in *Sleep*: Second Try

Optional

```
sleep (void* event, int sleep_priority)
{
        struct proc *p = curproc;
        int s;

        s = splhigh();
        lock spinlock;

        p->p_wchan = event;                     /* what are we waiting for */
        p->p_priority -> priority;              /* wakeup scheduler priority */
        p->p_stat = SSLEEP;                     /* transition curproc to sleep state */
        INSERTQ(&slpque[HASH(event)], p);               /* fiddle sleep queue */

        unlock spinlock;
        splx(s);

        mi_switch();                /* context switch */
        /* we're back */
}
```

Grab spinlock *and* disable interrupts.

Review: Threads vs. Processes

1. The *process* is a *kernel abstraction* for an independent executing program.

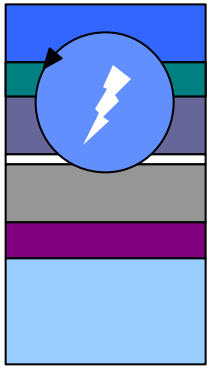
   includes at least one "thread of control"

   also includes a private address space (VAS)

   - VAS requires OS kernel support

   often the unit of resource ownership in kernel

   - e.g., memory, open files, CPU usage



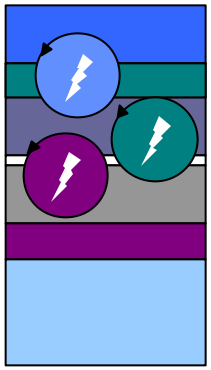
2. Threads may share an address space.

   Threads have "context" just like vanilla processes.

   - *thread context switch* vs. *process context switch*

   Every thread must exist within some process VAS.

   Processes may be "multithreaded" with thread primitives supported by a library or the kernel.



Implementing Processes: Questions

A process is an execution of a program within a private virtual address space (VAS).

1. What are the system calls to operate on processes?
2. How does the kernel maintain the state of a process?

   Processes are the “basic unit of resource grouping”.

3. How is the process virtual address space laid out?

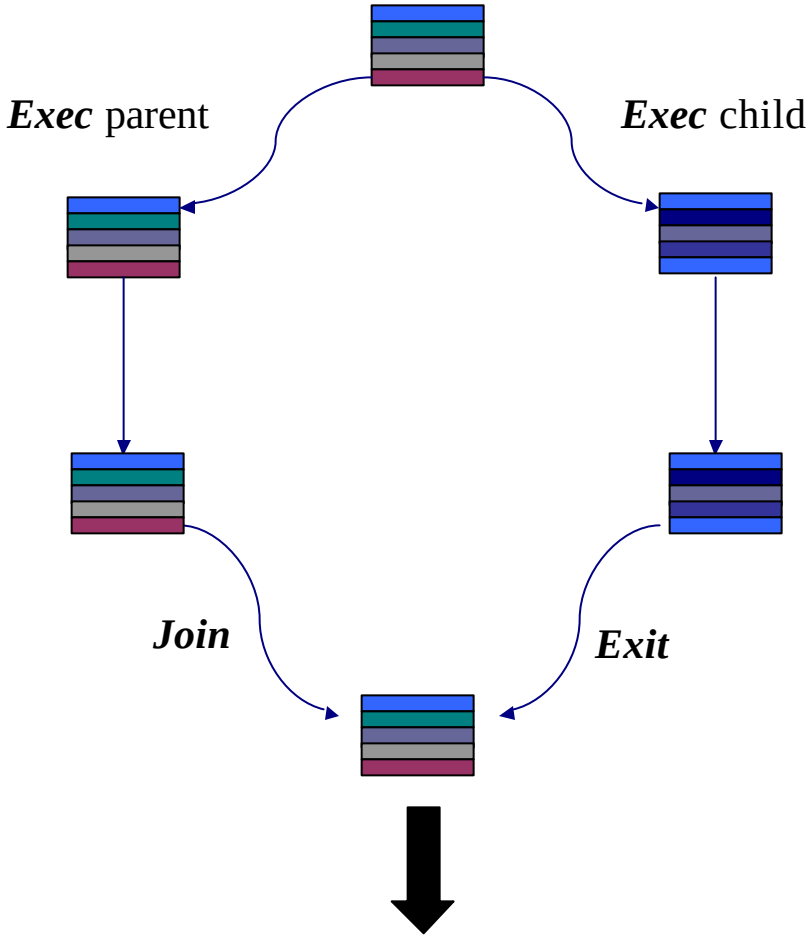
   What is the relationship between the program and the process?

4. How does the kernel create a new process?

   How to allocate physical memory for processes?

   How to create/initialize the virtual address space?

Nachos Exec/Exit/Join Example



SpaceID pid = Exec("myprogram", 0);

*Create a new process running the program "myprogram". Note: in Unix this is two separate system calls: **fork** to create the process and **exec** to execute the program.*

int status = Join(pid);

Called by the parent to wait for a child to exit, and "reap" its exit status. Note: child may have exited before parent calls Join!

Exit(status);

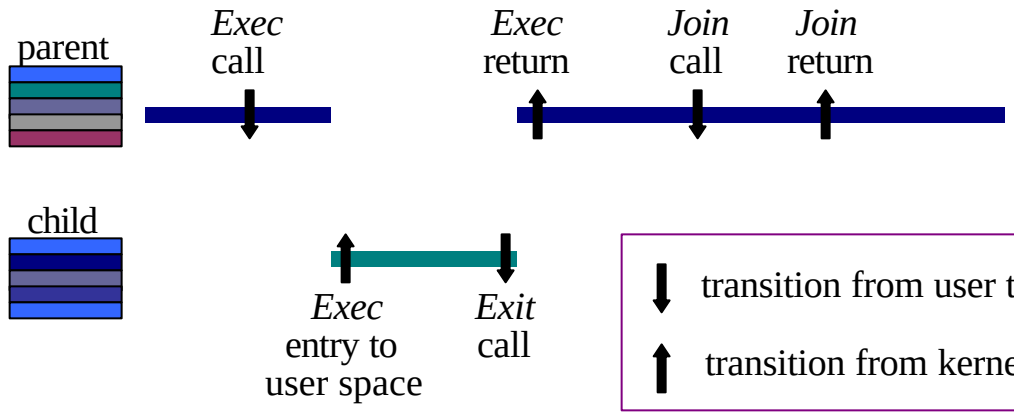
Exit with status, destroying process. Note: this is not the only way for a proess to exit!.

Mode Changes for Exec/Exit

Syscall traps and “returns” are not always paired.

- *Exec* “returns” (to child) from a trap that “never happened”
- *Exit* system call trap never returns
- system may switch processes between trap and return

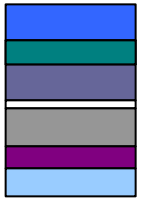
In contrast, interrupts and returns are strictly paired.



***Exec** enters the child by doctoring up a saved user context to “return” through.*

Process Internals

virtual address space

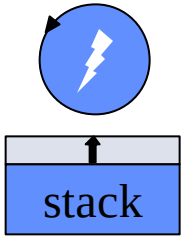


The address space is represented by *page table*, a set of translations to physical memory allocated from a kernel *memory manager*.

The kernel must initialize the process memory with the program image to run.

+

thread



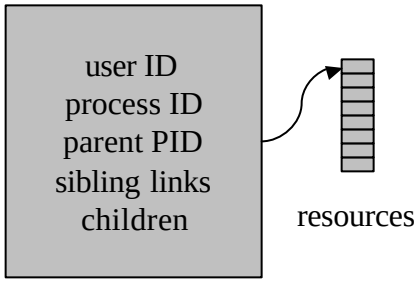
Each process has a thread bound to the VAS.

The thread has a saved user context as well as a system context.

The kernel can manipulate the user context to start the thread in user mode wherever it wants.

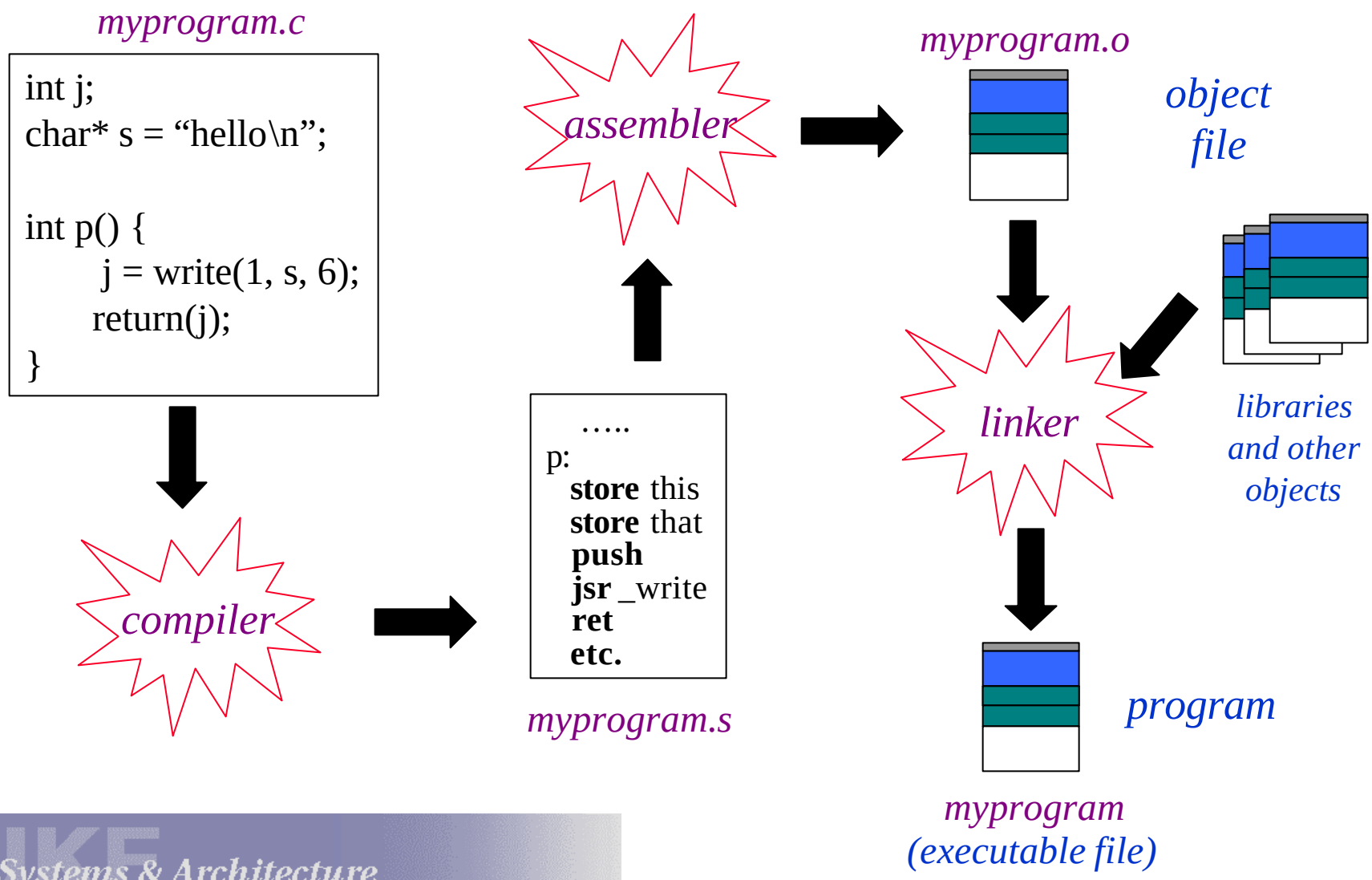
+

process descriptor

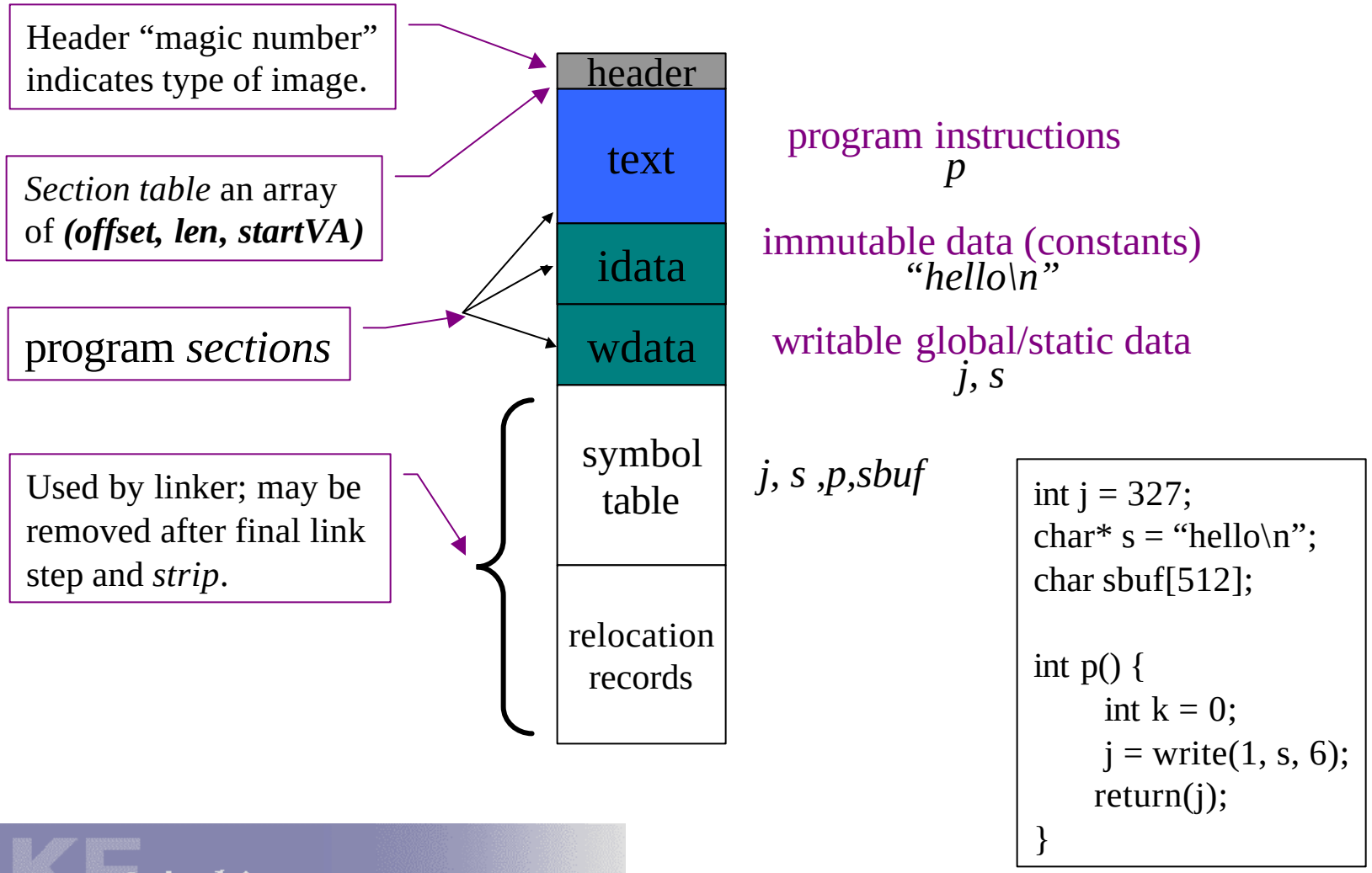


Process state includes a file descriptor table, links to maintain the process tree, and a place to store the exit status.

The Birth of a Program



What's in an Object File or Executable?

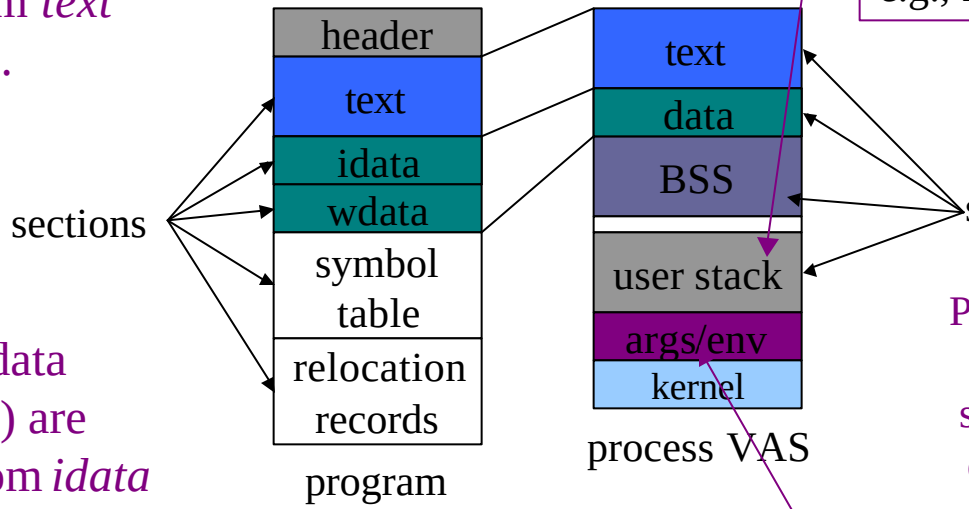



```
int j = 327;
char* s = "hello\n";
char sbuf[512];

int p() {
        int k = 0;
        j = write(1, s, 6);
       return(j);
}
```

The Program and the Process VAS

Process *text* segment is initialized directly from program *text* section.



BSS
"Block Started by Symbol"
(uninitialized global data)
e.g., heap and *sbuf* go here.

Process data segment(s) are initialized from *idata* and *wdata* sections.

Process BSS segment may be expanded at runtime with a system call (e.g., Unix sbrk) called by the heap manager routines.

Text and *idata* segments may be write-protected.

Process stack and BSS (e.g., heap) segment(s) are zero-filled.

Args/env strings copied in by kernel when the process is created.

Nachos: A Peek Under the Hood