CSC 369

Operating Systems

System Calls

Threads

Intro to Synchronization (maybe)

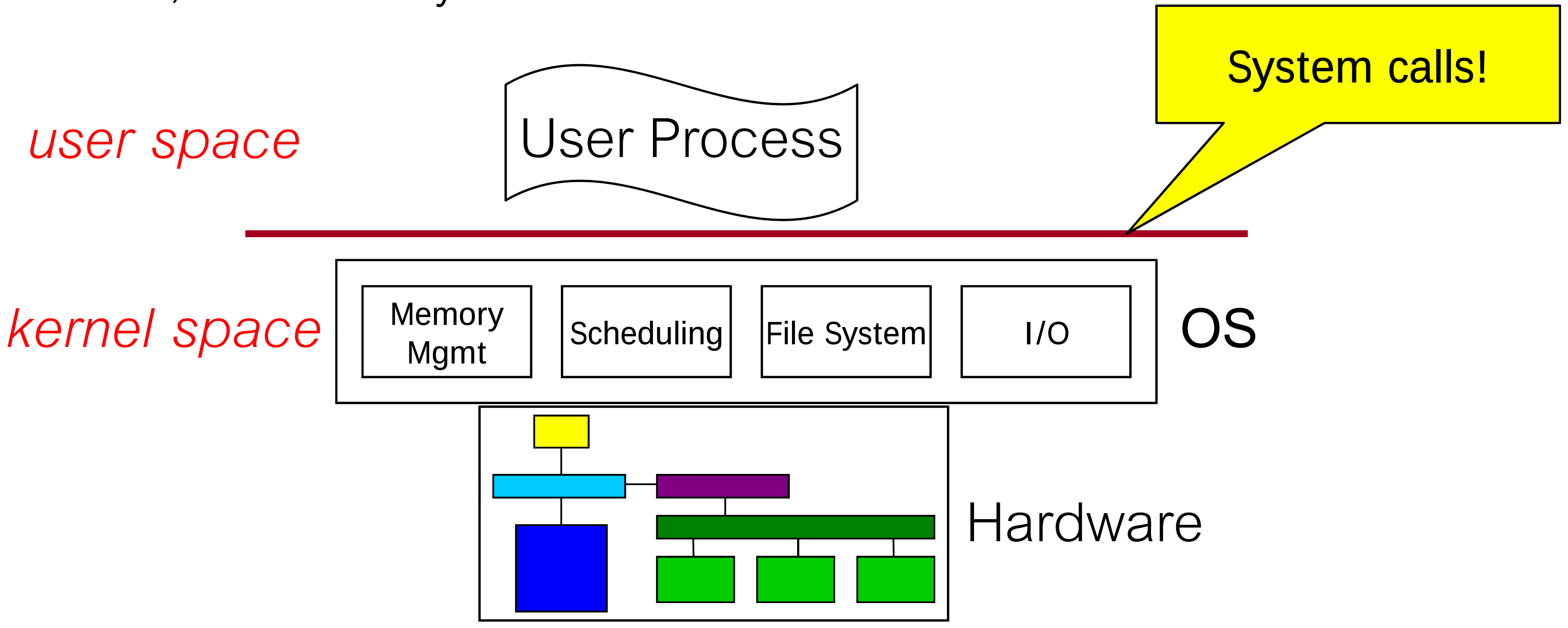
University of Toronto, Department of Computer Science

Today

- System Calls
- Threads
- Synchronization

Requesting OS Services

- Operating System and user programs are isolated from each other
- But OS provides service to user programs…
- So, how do they communicate?

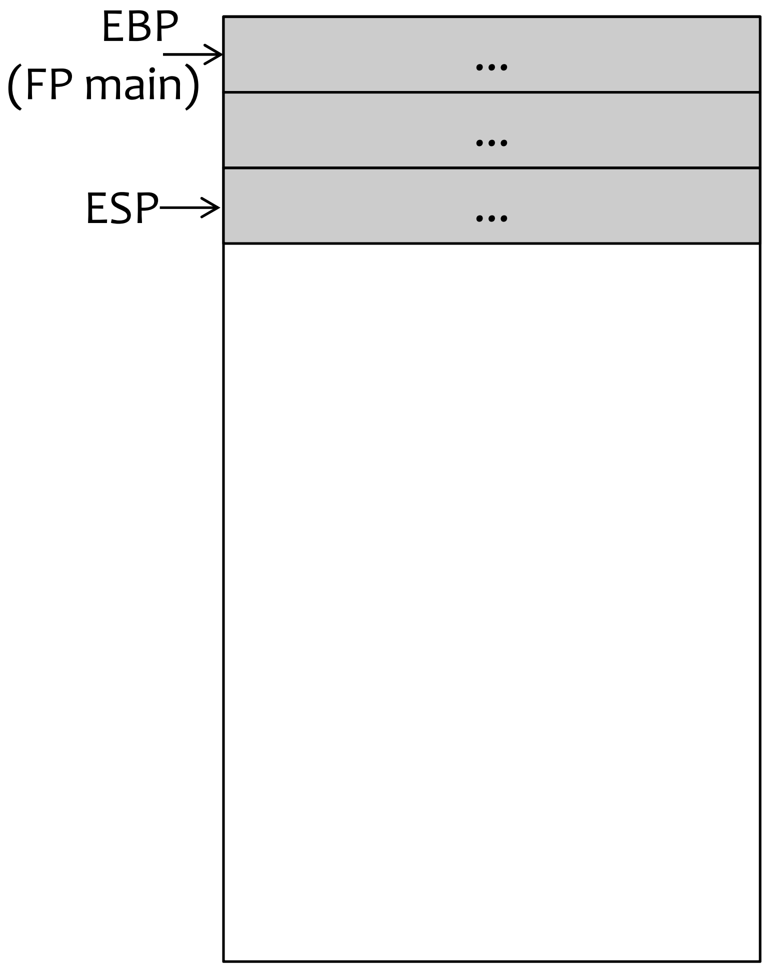


But first, function calls

- Suppose you are in assembly and about to make a function call
- C code below:

```
#include <stdio.h>
int pinkbunny(int x, int y) {
    int i = 10, j = 5;
    return x + y + i + j;
}

int main() {
    int r = 2;
    int q = 3;
    int result = pinkbunny(r, q);
    printf("result: %d\n", result);
    return 0;
}
```

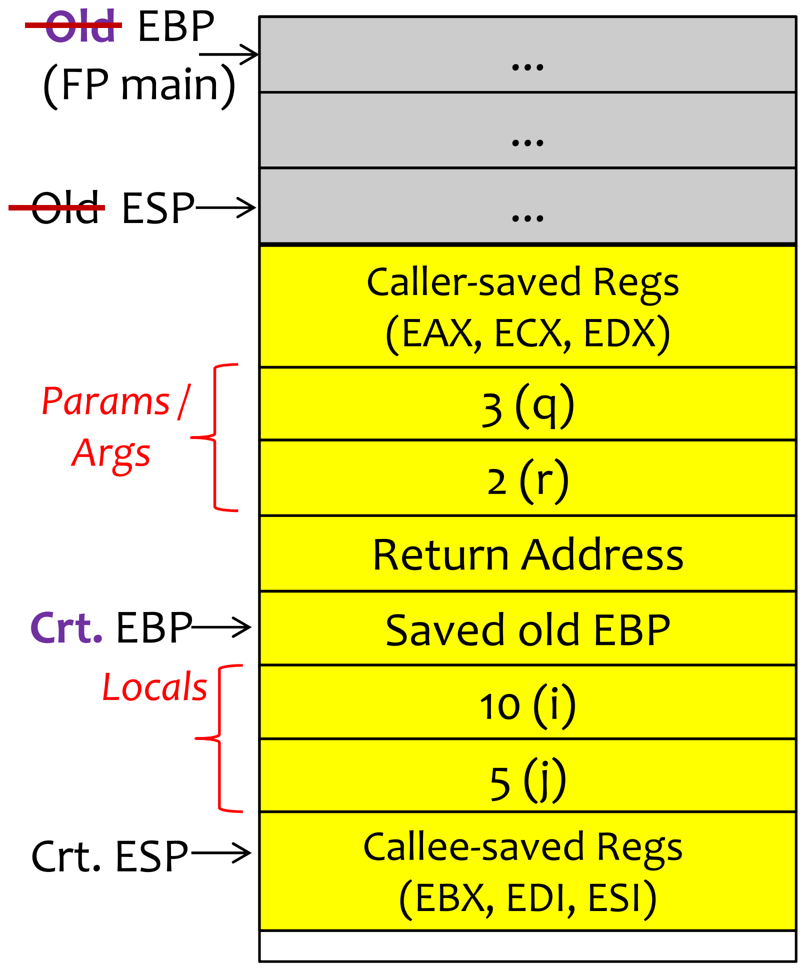


But first, function calls

- Suppose you are in assembly and about to make a function call
- C code below:

```
#include <stdio.h>
int pinkbunny(int x, int y) {
    int i = 10, j = 5;
    return x + y + i + j;
}

int main() {
    int r = 2;
    int q = 3;
    int result = pinkbunny(r, q);
    printf("result: %d\n", result);
    return 0;
}
```



But first, function calls

- Suppose you are in assembly and about to make a function call
- C code below:

```
#include <stdio.h>
int pinkbunny(int x, int y) {
    int i = 10, j = 5;
    return x + y + i + j;
}

int main() {
    int r = 2;
    int q = 3;
    int result = pinkbunny(r, q);
    printf("result: %d\n", result);
    return 0;
}
```

4.push frame pointer (EBP)
5.push callee registers
6.decr. stack (add locals)

7.put retval in EAX register
8.restore registers
9.ESP <- EBP (or incr stack)
10.pop EBP
11.ret

1.push registers on stack
2.push args on stack
3.call pinkbunny (push RA)

12.remove args from stack
13.Move EAX into 'result'
14.restore caller registers

X86 Calling a Function

To make a subroutine call, the caller should:

1. Before calling a subroutine, the caller should save the contents of certain registers that are designated *caller-saved*. The caller-saved registers are EAX, ECX, EDX. Since the called subroutine is allowed to modify these registers, if the caller relies on their values after the subroutine returns, the caller must push the values in these registers onto the stack (so they can be restored after the subroutine returns.
2. To pass parameters to the subroutine, push them onto the stack before the call. The parameters should be pushed in inverted order (i.e. last parameter first). Since the stack grows down, the first parameter will be stored at the lowest address (this inversion of parameters was historically used to allow functions to be passed a variable number of parameters).
3. To call the subroutine, use the call instruction. This instruction places the return address on top of the parameters on the stack, and branches to the subroutine code. This invokes the subroutine, which should follow the callee rules below.

After the subroutine returns (immediately following the call instruction), the caller can expect to find the return value of the subroutine in the register EAX. To restore the machine state, the caller should:

1. Remove the parameters from stack. This restores the stack to its state before the call was performed.
2. Restore the contents of caller-saved registers (EAX, ECX, EDX) by popping them off of the stack. The caller can assume that no other registers were modified by the subroutine.

Src: http://www.cs.virginia.edu/~evans/cs216/guides/x86.html

What is a System Call?

- “system call” == “a function call that invokes the operating system”
- Whenever an application wants to use a resource that the OS manages, it asks permission!
- How do you actually invoke the operating system?
- How do we keep applications from just using a resource *without* asking permission?

Interrupts

- Can be caused by hardware or software
- Interrupts signal CPU that a hardware device has an event that needs attention
  - E.g. Disk I/O completes, etc.
- Interrupts signal errors or requests for OS intervention (a system call)
  - Often called an “exception” or “trap”
- CPU jumps to a pre-defined routine (the interrupt handler)
- An OS is an event-driven program
  - The OS “responds” to requests

Boundary Crossings

- Getting to kernel mode
  - Explicit system call – request for service by application
  - Hardware interrupt
  - Software trap or exception
  - → Hardware has table of "Interrupt service routines"
  - → What should it save first?
- Kernel to user
  - When the OS is finished its task, get back to application
  - → OS sets up registers, MMU, mode for application
  - → Jumps to next application instruction

Enforcing restrictions

- Hardware runs in user mode or system mode
- Some instructions are privileged instructions: they can only run in system mode
- On a “system call interrupt”, the mode bit is switched to allow privileged instructions to occur
- What instructions/operations would you expect to be privileged?

Privileged instructions

- Access I/O device
  - Poll for IO, perform DMA, catch hardware interrupt
- Manipulate memory management
  - Set up page tables, load/flush the TLB and CPU caches (we'll see this later), etc.
- Configure various "mode bits"
  - Interrupt priority level, software trap vectors, etc.
- Call halt instruction
  - Put CPU into low-power or idle state until next interrupt
- These are enforced by the CPU hardware itself
  - Reminder: CPU has at least 2 protection levels: Kernel and user mode (x86: 4 rings)
  - CPU checks current protection level on each instruction!
  - What happens if user program tries to execute a privileged instruction?

How does the OS know what to do?

- When an interrupt occurs, there must be a reason
- The reason is stored in a register, and that register is used to invoke a handler
- A handler is just a function: remember signal handlers in 209?

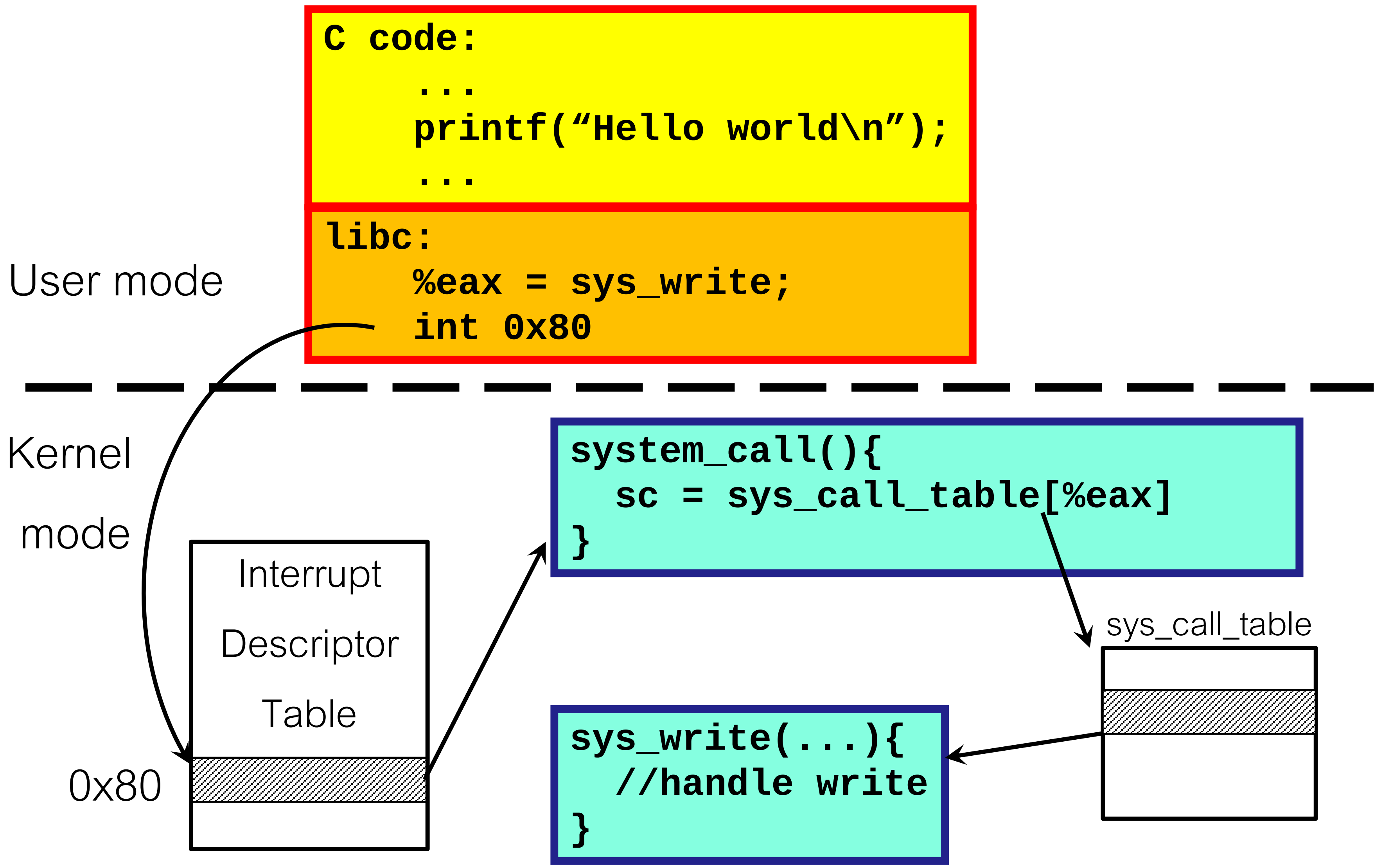
How to View the System Call Interface

- User program calls C library function with arguments
- C library function passes arguments to OS
  - Includes a system call identifier!
- Executes special instruction (x86: INT) to trap to system mode
  - Interrupt/trap vector transfers control to a handler routine
- Syscall handler figures out which system call is needed and calls a routine for that operation
- How does this differ from a normal C language function call? Why is it done this way?

System Call Operation

- Kernel must verify arguments that it is passed
  - Why?
- A fixed number of arguments can be passed in registers
  - Often pass the address of a user buffer containing data (e.g., for write())
  - Kernel must copy data from user space into its own buffers
- Result of system call is returned in register EAX

Example: Linux "write" system call



User mode

```
C code:
    ...
    printf("Hello world\n");
    ...
```

```
libc:
    %eax = sys_write;
    int 0x80
```

Kernel mode

```
system_call(){
  sc = sys_call_table[%eax]
}
```

```
sys_write(...){
  //handle write
}
```

System calls in Linux

- Macro defined as follows (X = number of params)
  - SYSCALL_DEFINEX(name, arg1type, arg1name, ..)

- Example: the write system call (number 4 in the sys_call_table)
  - In C (libc): long write(unsigned int fd, const char* buf, size_t count);
  - Kernel: SYSCALL_DEFINE3(write, unsigned int, fd, const char __user *, buf, size_t, count)

    (see in VM: /usr/src/linux-source-2.6.32/fs/read_write.c)
  - Naming: asmlinkage long sys_write(unsigned int fd, const char __user *buf, size_t count);

    (see in VM: /usr/src/linux-source-2.6.32/include/linux/syscalls.h)

System calls in Linux

- Can invoke any system call from userspace using the syscall() function
  - `syscall(syscall_no, arg1, arg2, arg3, ..)`
  - E.g., `const char msg[] = "Hello World!";`

    ```
    syscall(4, STDOUT_FILENO, msg, sizeof(msg)-1);
    ```
  - Equivalent to: `write(STDOUT_FILENO, msg, sizeof(msg)-1);`

- Tracing system calls:
  - Powerful mechanism to trace system call execution for an application
  - Use the strace command
  - The ptrace() system call is used to implement strace (also used by gdb)
  - Library calls can be traced using ltrace command

System calls dispatch

- Why do we need a system call table?
  - How would you get to the right routine?
  - If-then-else for each system call number?
  - Too inefficient!
- A system call is identified by a unique number
  - The system call number is passed into register %eax
  - Offers an index into an array of function pointers: the system call table!
- System call table: sys_call_table[__NR_syscalls] (approximately 300)
  - See all syscalls in VM: /usr/src/linux-source-2.6.32/arch/x86/kernel/syscall_table_32.S

```
...
.long  sys_read
.long  sys_write
.long  sys_open
```

System call dispatch

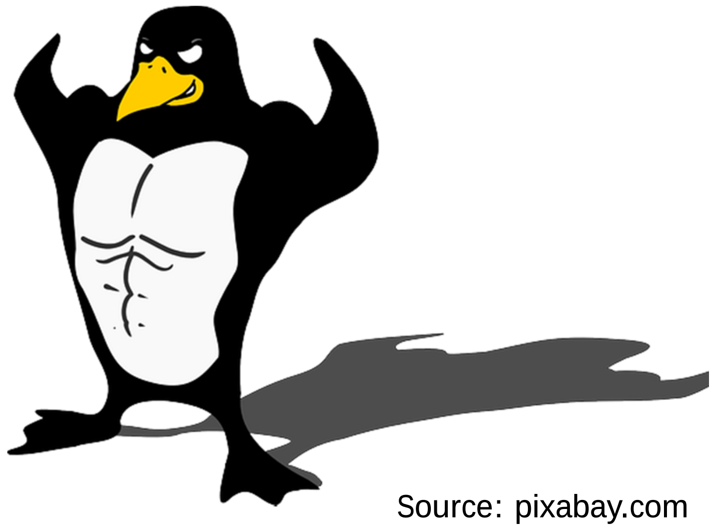
1. Kernel assigns each system call type a system call number
2. Kernel initializes system call table, mapping each system call number to a function implementing that system call
3. User process sets up system call number and arguments
4. User process runs *int N* (on Linux, N=80h)
5. Hardware switches to kernel mode and invokes kernel's interrupt handler for X (interrupt dispatch)
6. Kernel looks up syscall table using system call number
7. Kernel invokes the corresponding function
8. Kernel returns by running *iret* (interrupt return)

Passing system call parameters

- The first parameter is always the syscall number
    - Stored in eax
- Can pass up to 6 parameters:
    - ebx, ecx, edx, esi, edi, ebp
- If more than 6 parameters are needed, package the rest in a struct and pass a pointer to it as the 6th parameter
- *Problem:* must validate user pointers. Why? How?
- *Solution:* safe functions to access user pointers: copy_from_user(), copy_to_user(), etc.

Intel Fast System Calls

- Linux used to implement system calls using “int 0x80”
- Turned out too slow on Pentium IV+
- New shiny way to implement system calls!
    - Alternative mechanism leverages the SYSENTER/SYSEXIT instructions provided by the processor



Source: pixabay.com

Assignment 1

- System calls

Today

- System Calls
- Threads
- Synchronization

Process Cooperation

- A process is independent if it cannot affect or be affected by the other processes executing in the system
- No data sharing => process is independent

- A process is cooperating if it is not independent
- Cooperating processes must be able to communicate with each other and to synchronize their actions

Interprocess Communication

- Cooperating processes need to exchange information
  - Shared memory (e.g., fork(), shmget()/shmat(),..)
  - Message passing
- Message passing models
  - Direct
    - Send(P, msg) – send msg to process P
    - Receive(Q, msg) – receive msg from process Q
    - Pipes
  - Indirect
    - Send/receive reference ports (like mailboxes) rather than specific processes
  - Involves system calls & data copying => costly!

Threads: Concurrent Servers

- Recall our Web server example (revisit CSC209):

```
while (1) {
  int sock = accept();
  if ((child_pid = fork()) == 0) {
    Handle client request
  } else {
    Close socket
  }
}
```

- Is this efficient?
  - Using fork() to create new processes to handle requests in parallel is overkill for such a simple task

Parallel Programs

- To execute web server, or any parallel program using fork(), we need to
  - Create several processes that execute in parallel
  - Cause each to map to the same address space to share data
    - They are all part of the same computation
  - Have the OS schedule these processes in parallel (logically or physically)
- This situation is very inefficient
  - Space: PCB, page tables, etc.
  - Time: create data structures, fork and copy addr space, etc.
  - Inter-process communication (IPC): extra work is needed to share and communicate across isolated processes

Threads: Concurrent Servers

- Instead, we can create a new thread for each request

```
web_server() {
   while (1) {
    int sock = accept();
          thread_fork(handle_request, sock);
   }
}

handle_request(int sock) {
    Process request
    close(sock);
}
```

From Processes to Threads

- Recall that a process includes:
  - Address space
    - Code
    - Heap
  - Execution state
    - Program counter
    - Stack and stack pointer
  - OS resources
    - Open file table, etc.

Key idea

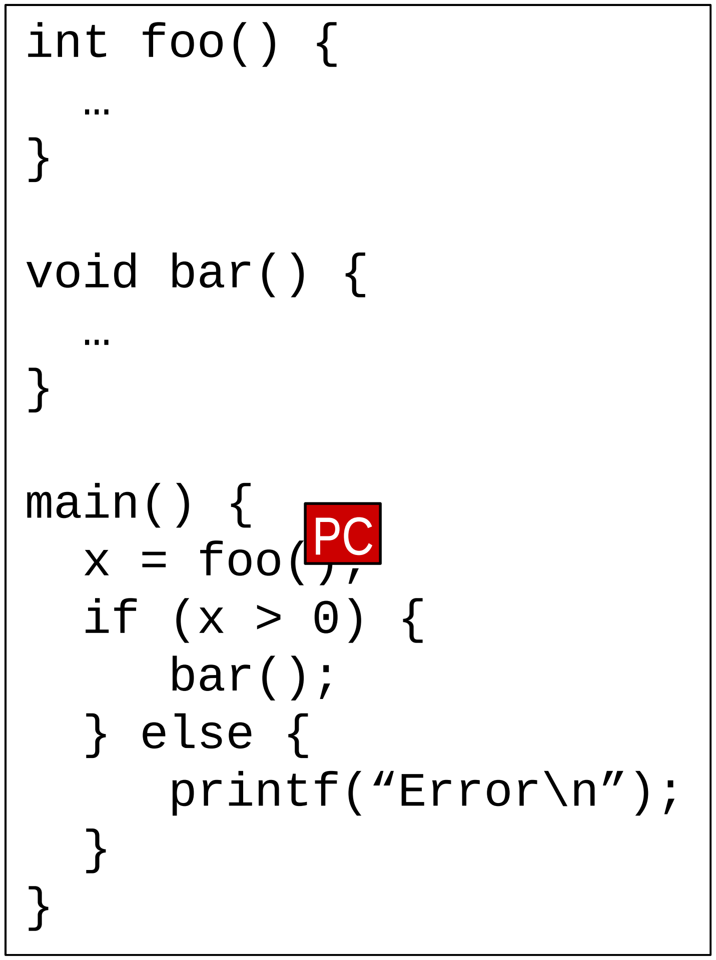
- Separate the address space from the execution state
- Then multiple “threads of execution” can execute in a single address space

Why?

- Sharing:
  - Threads can solve a single problem concurrently and can easily share code, heap, and global variables
- Lighter weight:
  - Faster to create and destroy
  - Potentially faster context switch times
- Concurrent programming performance gains:
  - Overlapping computation and I/O

What is a Thread?

- A thread is a single *control flow through a program*
  - What is a "control flow?
  - How is it represented?
- A program with multiple control flows is *multithreaded*
  - OS must interact with multiple running programs, so it is naturally multithreaded

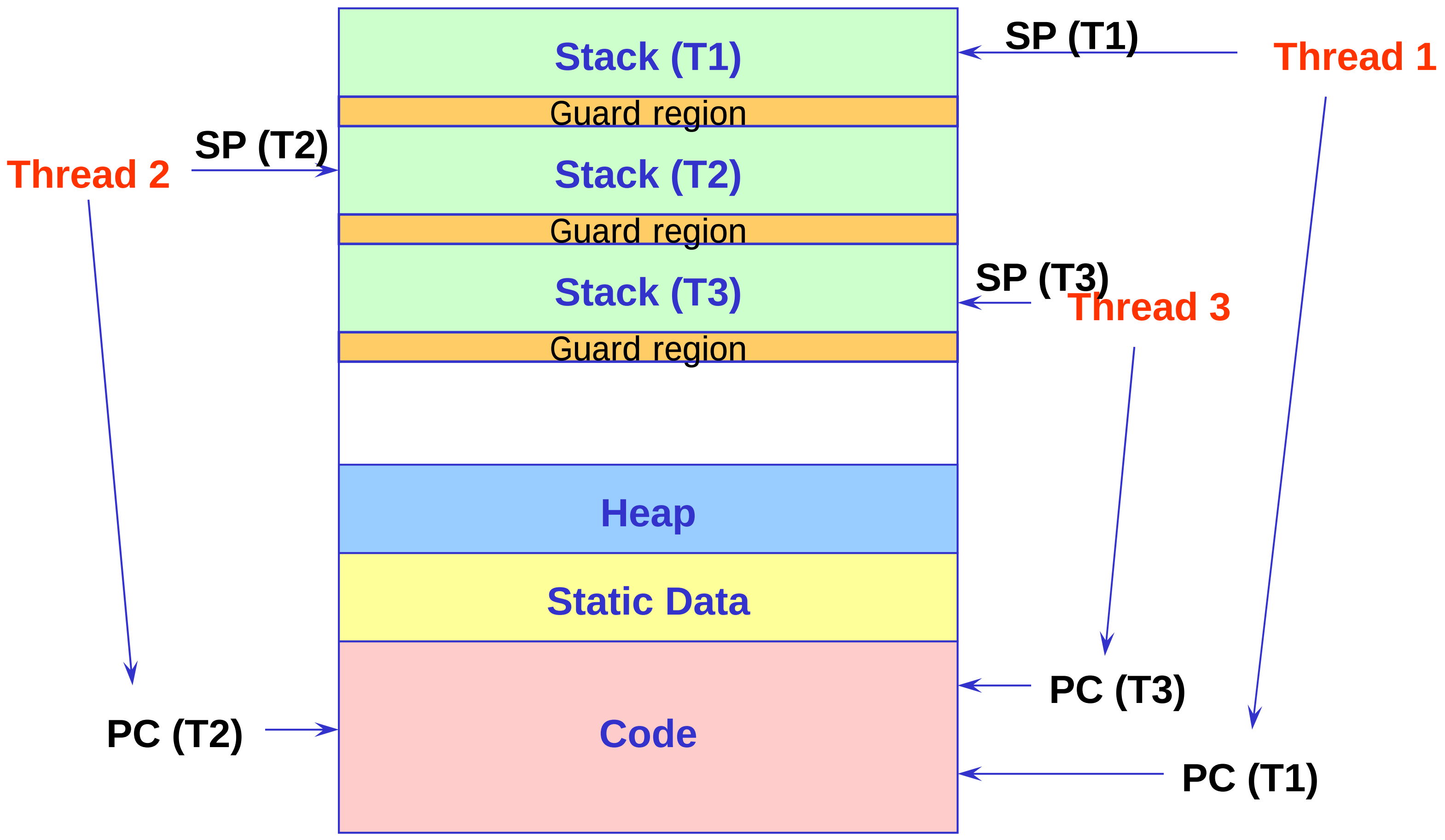


```
int foo() {
  …
}

void bar() {
  …
}

main() {
  x = foo();
  if (x > 0) {
      bar();
  } else {
      printf("Error\n");
  }
}
```

Multithreaded Process Address Space



Kernel-Level Threads

- Modern OSs have taken the execution aspect of a process and separated it out into thread abstraction
  - To make concurrency cheaper
- The OS now manages threads and processes
  - All thread operations are implemented in the kernel
  - The OS schedules all of the threads in the system
- Called kernel-level threads or lightweight processes
  - Linux: tasks NT: threads   Solaris: lightweight processes (LWP)

Kernel Thread Limitations

- Kernel-level threads make concurrency much cheaper than processes
  - Much less state to allocate and initialize
- However, for fine-grained concurrency, kernel-level threads may suffer from too much overhead
  - Thread operations still require system calls
    - Ideally, want thread operations to be as fast as a procedure call
  - Kernel-level threads have to be general to support the needs of all programmers, languages, runtimes, etc.
- For fine-grained concurrency, need even “cheaper” threads

User-Level Threads

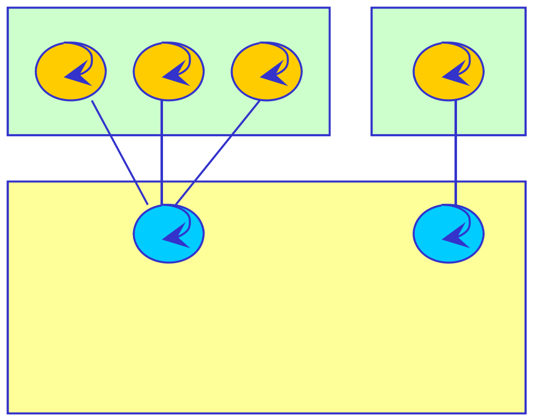
- To make threads cheap and fast, need to be implemented at user level
  - Kernel-level threads are managed by the OS
  - User-level threads are managed entirely by the run-time system (user-level library)
- User-level threads are small and fast
  - A thread is simply represented by a PC, registers, stack, and small thread control block (TCB)
  - Creating a new thread, switching between threads, and synchronizing threads are done via procedure call (no kernel involvement)
  - User-level thread operations are up to 100x faster than kernel threads
    - But this depends on the quality of both implementations!

U/L Thread Limitations

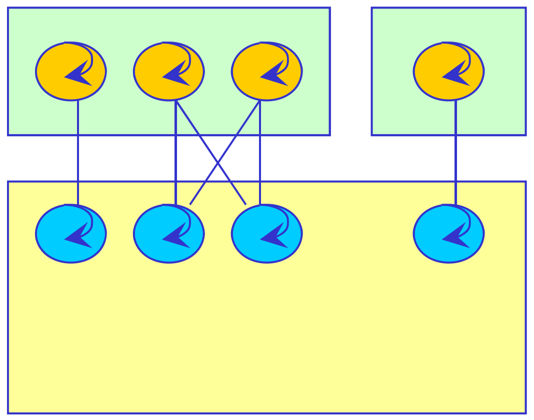
- User-level threads are not a perfect solution
  - As with everything else, they are a tradeoff
- User-level threads are invisible to the OS
  - They are not well integrated with the OS
- As a result, the OS can make poor decisions
  - Scheduling a process with only idle threads
  - Blocking a process whose thread initiated an I/O, even though the process has other threads that can execute
  - De-scheduling a process with a thread holding a lock
- Solving this requires communication between the kernel and the user-level thread manager

Hybrid Kernel and User Threads

- Another possibility is to use both kernel and user-level threads
  - Can associate a user-level thread with a kernel-level thread
  - Or, multiplex user-level threads on top of kernel-level threads



Multiplexing user-level threads on a single kernel thread for each process



Multiplexing user-level threads on multiple kernel threads for each process

POSIX threads

- Standardized C language threads programming API
- Known as *pthreads*
- Specifies interface, not implementation!
  - Implementation may be kernel-level threads, user-level threads, or hybrid
  - Linux pthreads implementation uses kernel-level threads

- Excellent tutorial:
  - https://computing.llnl.gov/tutorials/pthreads/

Examples and take-aways

- sequential vs. parallel, pthreads vs. processes ...
- Bottomline – in general:
  - processes are more heavy-weight than threads and have a higher startup/shutdown cost (..not by much in Linux though)
  - processes are safer and more secure (each process has its own address space)
    - a thread crash takes down all other threads
    - a thread's buffer overrun creates security problem for all
  - which one to use? depends on application type!

Next Time…

- Intro to Concurrency and Synchronization
  - Critical section problem
  - Synchronization Hardware
  - Locks
  - Condition Variables