

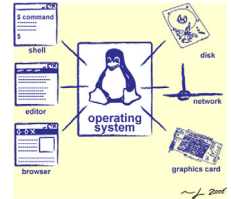
CS162
Operating Systems and
Systems Programming
Lecture 3

Abstractions 1: Threads and Processes
A quick, programmer's viewpoint

January 23rd, 2024
Prof. John Kubitowicz
<http://cs162.eecs.Berkeley.edu>

Goals for Today: The Thread Abstraction

- **What** threads are
 - And what they are not
- **Why** threads are useful (motivation)
- **How** to write a program using threads
- **Alternatives** to using threads



1/23/24

Kubitowicz CS162 © UCB Spring 2024

Lec 3.2

Recall: Four Fundamental OS Concepts

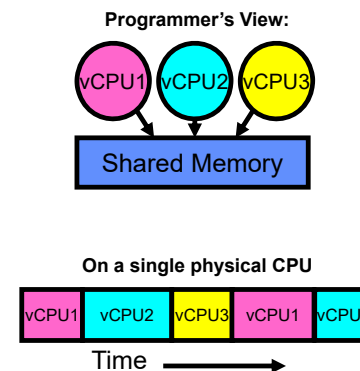
- **Thread: Execution Context**
 - Fully describes program state
 - Program Counter, Registers, Execution Flags, Stack
- **Address space** (with or w/o translation)
 - Set of memory addresses accessible to program (for read or write)
 - May be distinct from memory space of the physical machine (in which case programs operate in a virtual address space)
- **Process: an instance of a running program**
 - Protected Address Space + One or more Threads
- **Dual mode operation / Protection**
 - Only the “system” has the ability to access certain resources
 - Combined with translation, isolates programs from each other and the OS from programs

1/23/24

Kubitowicz CS162 © UCB Spring 2024

Lec 3.3

Recall: Illusion of Multiple Processors



- Threads are *virtual cores*
- Multiple threads: *Multiplex* hardware in time
- A Thread is *executing* on a processor when it is resident in that processor's registers
- Each virtual core (thread) has:
 - Program counter (PC), stack pointer (SP)
 - Registers – both integer and floating point
- Where is “it” (the thread)?
 - On the real (physical) core, or
 - Saved in chunk of memory – called the *Thread Control Block (TCB)*

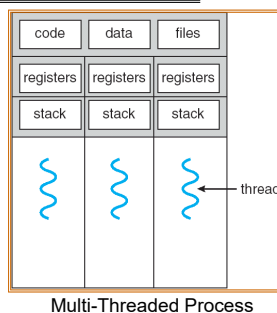
1/23/24

Kubitowicz CS162 © UCB Spring 2024

Lec 3.4

Recall: Process

- **Definition: execution environment with Restricted Rights**
 - One or more threads in a (protected) Address Space
 - Owns memory (address space), file descriptors, network connections, ...
- Instance of a running program
 - When you run an executable, it runs in its own process
 - Application: one or more processes working together
- Why processes?
 - Protected from each other!
 - OS Protected from them
- In modern OS, anything that runs outside of the kernel runs in a process

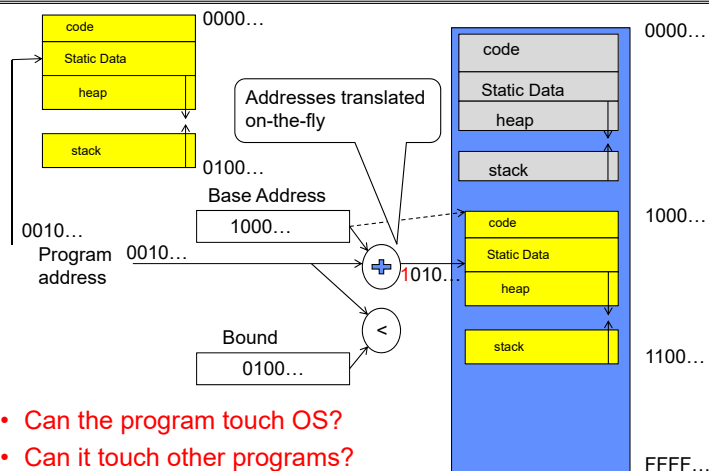


1/23/24

Kubiatowicz CS162 © UCB Spring 2024

Lec 3.5

Recall: Simple address translation with Base and Bound



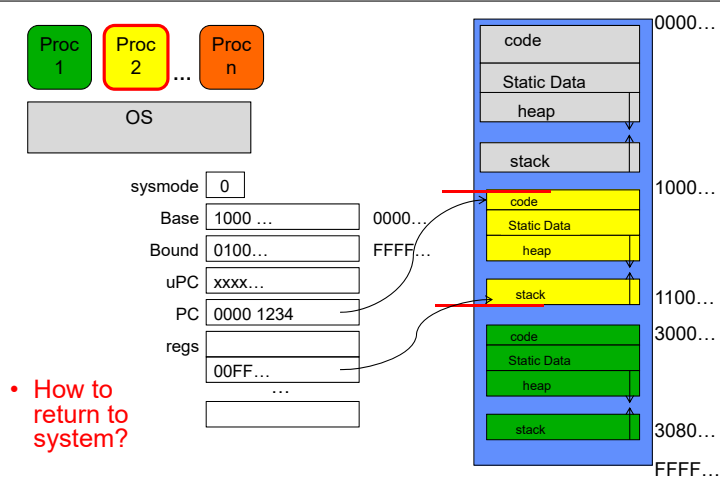
- Can the program touch OS?
- Can it touch other programs?

1/23/24

Kubiatowicz CS162 © UCB Spring 2024

Lec 3.6

Simple B&B: User => Kernel



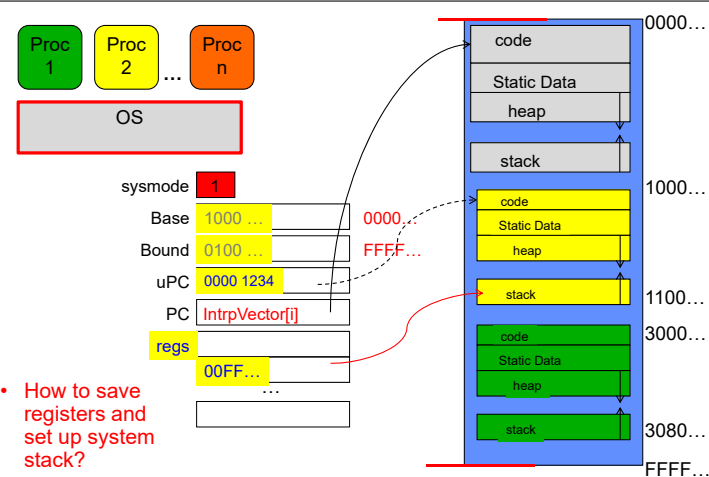
- How to return to system?

1/23/24

Kubiatowicz CS162 © UCB Spring 2024

Lec 3.7

Simple B&B: Interrupt



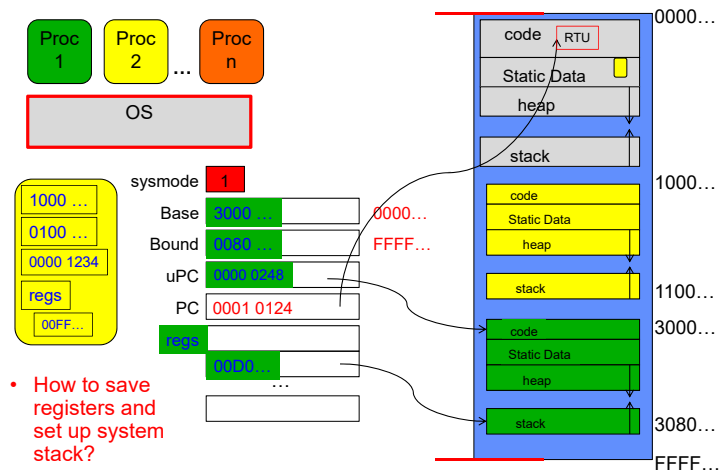
- How to save registers and set up system stack?

1/23/24

Kubiatowicz CS162 © UCB Spring 2024

Lec 3.8

Simple B&B: Switch User Process

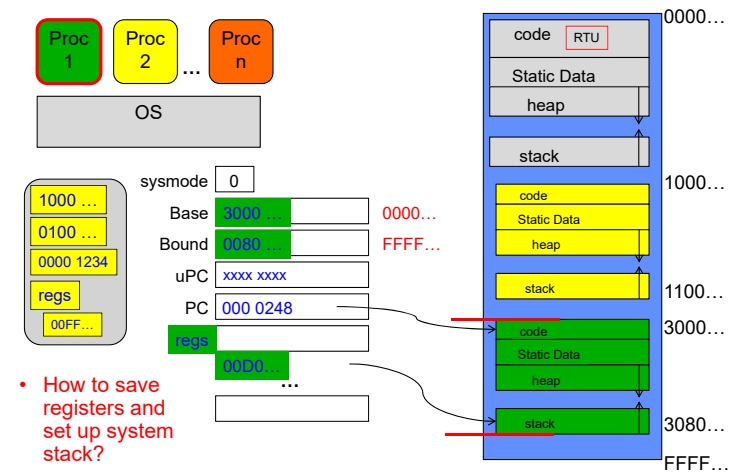


1/23/24

Kubiatowicz CS162 © UCB Spring 2024

Lec 3.9

Simple B&B: “resume”



1/23/24

Kubiatowicz CS162 © UCB Spring 2024

Lec 3.10

Is Simple Base and Bound Enough for General Systems?

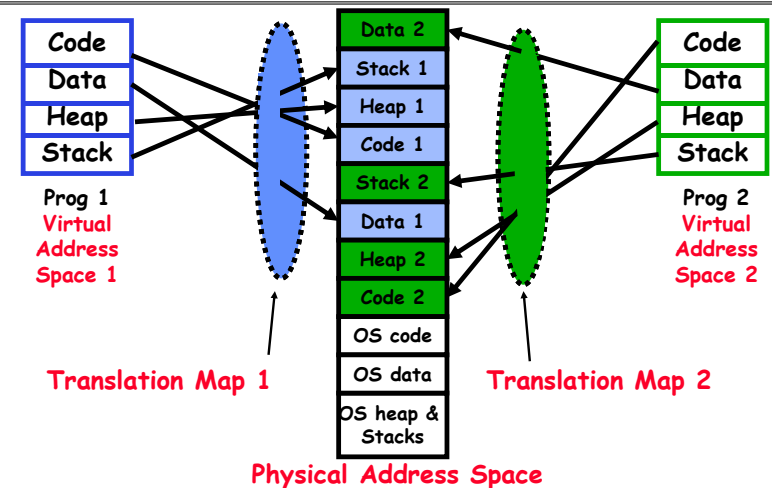
- **NO: Too simplistic for real systems**
- Inflexible/Wasteful:
 - Must dedicate physical memory for *potential* future use
 - (Think stack and heap!)
- Fragmentation:
 - Kernel has to somehow fit whole processes into contiguous block of memory
 - After a while, memory becomes fragmented!
- Sharing:
 - Very hard to share any data between Processes or between Process and Kernel
 - Need to communicate indirectly through the kernel...

1/23/24

Kubiatowicz CS162 © UCB Spring 2024

Lec 3.11

Better: Translation through Page Table (More soon!)



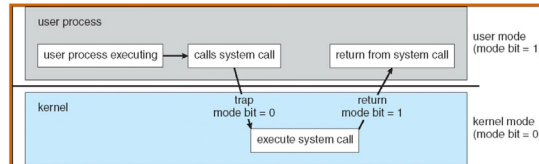
1/23/24

Kubiatowicz CS162 © UCB Spring 2024

Lec 3.12

Recall: Dual Mode Operation

- Processes (i.e., programs you run) execute in **user mode**
 - To perform privileged actions, processes request services from the OS kernel
 - Carefully controlled transition from user to kernel mode
- Kernel executes in **kernel mode**
 - Performs privileged actions to support running processes
 - ... and configures hardware to properly protect them (e.g., address translation)
- Carefully controlled transitions between user mode and kernel mode
 - System calls, interrupts, exceptions

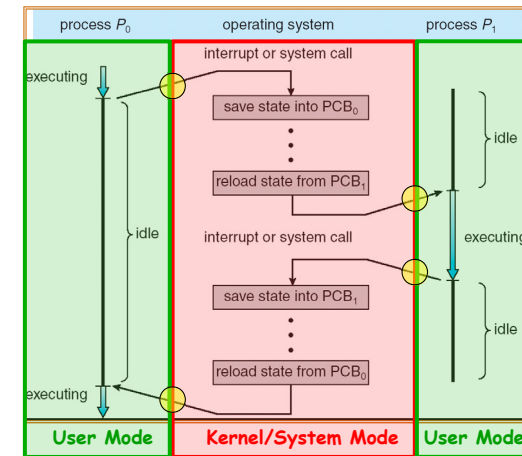


1/23/24

Kubiatowicz CS162 © UCB Spring 2024

Lec 3.13

Adding Protection: CPU Switch From Process A to Process B



1/23/24

Kubiatowicz CS162 © UCB Spring 2024

Lec 3.14

Running Many Programs

- We have the basic *mechanism* to:
 - switch between user processes and the kernel,
 - the kernel can switch among user processes,
 - Protect OS from user processes and processes from each other
- Questions:
 - How do we represent user processes in the OS?
 - How do we decide which user process to run?
 - How do we pack up the process and set it aside?
 - How do we get a stack and heap for the kernel?
 - Aren't we wasting a lot of memory?

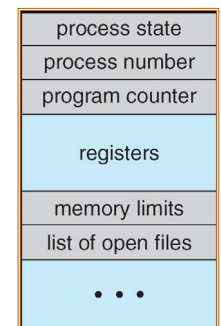
1/23/24

Kubiatowicz CS162 © UCB Spring 2024

Lec 3.15

Multiplexing Processes: The Process Control Block

- Kernel represents each process with a process control block (PCB)
 - Status (running, ready, blocked, ...)
 - Register state (when not ready)
 - Process ID (PID), User, Executable, Priority, ...
 - Execution time, ...
 - Memory space, translation, ...
- Kernel *Scheduler* maintains a data structure containing the PCBs
 - Give out CPU to different processes
 - This is a Policy Decision
- Give out non-CPU resources
 - Memory/IO
 - Another policy decision



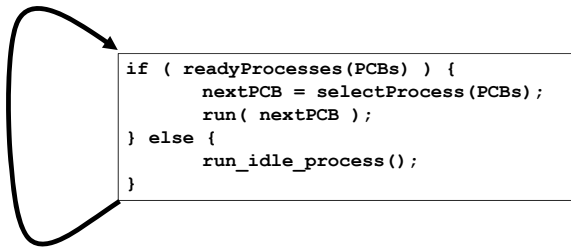
Process Control Block

1/23/24

Kubiatowicz CS162 © UCB Spring 2024

Lec 3.16

Scheduler



- Scheduling: Mechanism for deciding which processes/threads receive hardware CPU time, when, and for how long
- Lots of different scheduling policies provide ...
 - Fairness or
 - Realtime guarantees or
 - Latency optimization or ..

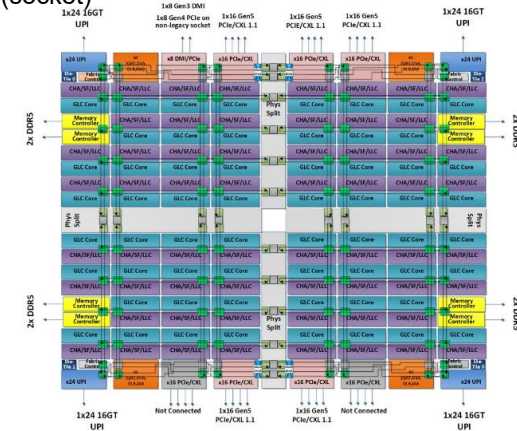
1/23/24

Kubiatowicz CS162 © UCB Spring 2024

Lec 3.17

Also Recall: The World Is Parallel: Sapphire Rapids (2023)

- Up to 60 cores, 120 threads/package (socket)
 - Up to 4 “chipllets” bonded together
- Network:
 - On-chip Mesh Interconnect
 - Fast off-chip network (UPI): directly connects 8-chips
 - 480 cores/shared memory domain!
- Each Core Has:
 - 80 KB L1 Cache
 - 2 MB L2 Cache
 - Fraction of up to 112.5 MB L3 Cache
- DRAM/chips
 - Up to 4 TiB of DDR5 memory
- Many Accelerators of different types
 - Graphics, Encryption, AI, Security



Sapphire Rapids 4-chiplet single package

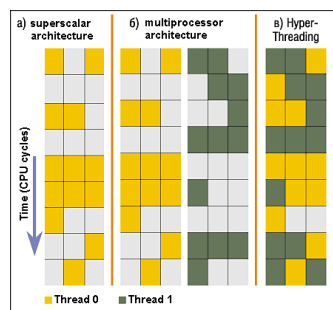
1/23/24

Kubiatowicz CS162 © UCB Spring 2024

Lec 3.18

Simultaneous MultiThreading/Hyperthreading

- Hardware scheduling technique
 - Avoids software overhead of multiplexing
 - Superscalar processors can execute multiple instructions that are independent.
 - Hyperthreading duplicates register state to make a second “thread,” allowing more instructions to run.
- Can schedule each thread as if were separate CPU
 - But, sub-linear speedup!
- Original technique called “Simultaneous Multithreading”
 - <http://www.cs.washington.edu/research/smt/index.html>
 - SPARC, Pentium 4/Xeon (“Hyperthreading”), Power 5



Colored blocks show instructions executed

1/23/24

Kubiatowicz CS162 © UCB Spring 2024

Lec 3.19

Administrivia: Getting started!

- Kubiatowicz Office Hours:
 - Tuesday/Thursday 3-4pm, in 673 Soda Hall
- Homework 0: **Due Tomorrow!**
 - Get familiar with the cs162 tools
 - configure your VM, submit via git
 - Practice finding out information:
 - » How to use GDB? How to understand output of unix tools?
 - » We don't assume that you already know everything!
 - » Learn to use “man” (command line), “help” (in gdb, etc), google
- Project 0: **Started Yesterday!**
 - Learn about Pintos and how to modify and debug kernel
 - Important for getting started on projects!
- Should be going to sections now – Important information there
 - Any section will do until groups assigned

1/23/24

Kubiatowicz CS162 © UCB Spring 2024

Lec 3.20

Administrivia (Con't)

- **THIS Friday is Drop Deadline! HARD TO DROP LATER!**
 - If you know you are going to drop, do so now to leave room for others on waitlist!
 - Why do we do this? So that groups aren't left without members!
- Group sign up via autograder form next week
 - Get finding groups of 4 people ASAP
 - Priority for same section; if cannot make this work, keep same TA
 - Remember: Your TA needs to see you in section!
- Midterm 1: 2/15
 - 8-10PM in person
 - We will say more about material when we get closer...
- Midterm 1 conflicts
 - We will handle these conflicts after have final class roster
 - Watch for queries by HeadTA to collect information

1/23/24

Kubiatowicz CS162 © UCB Spring 2024

Lec 3.21

CS 162 Collaboration Policy



- Explaining a concept to someone in another group
- Discussing algorithms/testing strategies with other groups
- Discussing debugging approaches with other groups
- Searching online for generic algorithms (e.g., hash table)



- Sharing code or test cases with another group or individual (including HW!)
- Copying OR reading another group's code or test cases
- Copying OR reading online code or test cases from prior years
- Helping someone in another group to debug their code

- We compare all project and HW submissions against prior year submissions and online solutions and will take actions (described on the course overview page) against offenders
- Don't put a friend in a bad position by asking for help that they shouldn't give!

1/23/24

Kubiatowicz CS162 © UCB Spring 2024

Lec 3.22

More About Threads: What are they?

- Definition from before: *A single unique execution context*
 - Describes its representation
- It provides the abstraction of: *A single execution sequence that represents a separately schedulable task*
 - Also a valid definition!
- Threads are a mechanism for *concurrency* (overlapping execution)
 - However, they can also run in *parallel* (simultaneous execution)
- Protection is an orthogonal concept
 - A protection domain can contain one thread or many

1/23/24

Kubiatowicz CS162 © UCB Spring 2024

Lec 3.23

Motivation for Threads

- Operating systems must handle multiple things at once (**MTAO**)
 - Processes, interrupts, background system maintenance
- Networked servers must handle MTAO
 - Multiple connections handled simultaneously
- Parallel programs must handle MTAO
 - To achieve better performance
- Programs with user interface often must handle MTAO
 - To achieve user responsiveness while doing computation
- Network and disk bound programs must handle MTAO
 - To hide network/disk latency
 - Sequence steps in access or communication

I made this term up!

1/23/24

Kubiatowicz CS162 © UCB Spring 2024

Lec 3.24

Threads Allow Handling MTAO

- Threads are a unit of *concurrency* provided by the OS
- Each thread can represent one thing or one task

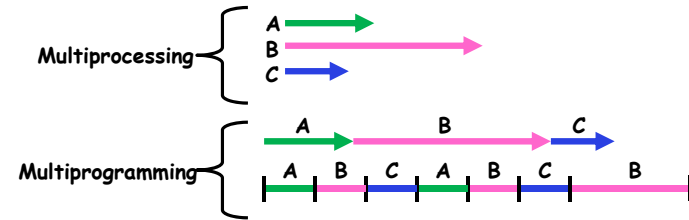
1/23/24

Kubiatowicz CS162 © UCB Spring 2024

Lec 3.25

Multiprocessing vs. Multiprogramming

- Some Definitions:
 - Multiprocessing: Multiple CPUs(cores)
 - Multiprogramming: Multiple jobs/processes
 - Multithreading: Multiple threads/processes
- What does it mean to run two threads concurrently?
 - Scheduler is free to run threads in any order and interleaving
 - Thread may run to completion or time-slice in big chunks or small chunks



1/23/24

Kubiatowicz CS162 © UCB Spring 2024

Lec 3.26

Concurrency is not Parallelism

- Concurrency is about handling multiple things at once (MTAO)
- Parallelism is about doing multiple things *simultaneously*
- Example: Two threads on a single-core system...
 - ... execute concurrently ...
 - ... but *not* in parallel
- Each thread handles or manages a separate thing or task...
- But those tasks are not necessarily executing simultaneously!

1/23/24

Kubiatowicz CS162 © UCB Spring 2024

Lec 3.27

Silly Example for Threads

- Imagine the following program:

```
main() {
    ComputePI("pi.txt");
    PrintClassList("classlist.txt");
}
```
- What is the behavior here?
- Program would never print out class list
- Why? ComputePI would never finish

1/23/24

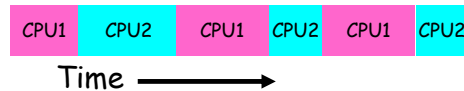
Kubiatowicz CS162 © UCB Spring 2024

Lec 3.28

Adding Threads

- Version of program with threads (loose syntax):


```
main() {
    create_thread(ComputePI, "pi.txt");
    create_thread(PrintClassList, "classlist.txt");
}
```
- `create_thread`: Spawns a new thread running the given procedure
 - Should* behave as if another CPU is running the given procedure
- Now, you would actually see the class list



More Practical Motivation: Compute/I/O overlap

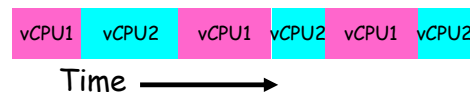
Back to Jeff Dean's
"Numbers Everyone
Should Know"

Handle I/O in
separate thread,
avoid blocking
other progress

L1 cache reference	0.5 ns
Branch mispredict	5 ns
L2 cache reference	7 ns
Mutex lock/unlock	25 ns
Main memory reference	100 ns
Compress 1K bytes with Zippy	3,000 ns
Send 2K bytes over 1 Gbps network	20,000 ns
Read 1 MB sequentially from memory	250,000 ns
Round trip within same datacenter	500,000 ns
Disk seek	10,000,000 ns
Read 1 MB sequentially from disk	20,000,000 ns
Send packet CA->Netherlands->CA	150,000,000 ns

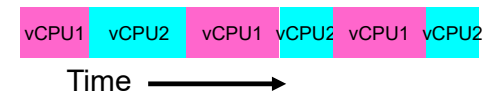
Threads Mask I/O Latency

- A thread is in one of the following three states:
 - RUNNING – running
 - READY – eligible to run, but not currently running
 - BLOCKED – ineligible to run
- If a thread is waiting for an I/O to finish, the OS marks it as BLOCKED
- Once the I/O finally finishes, the OS marks it as READY

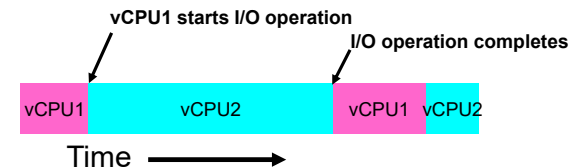


Threads Mask I/O Latency

- If no thread performs I/O:



- If thread 1 performs a blocking I/O operation:



A Better Example for Threads

- Version of program with threads (loose syntax):


```
main() {
    create_thread(ReadLargeFile, "pi.txt");
    create_thread(RenderUserInterface);
}
```
- What is the behavior here?
 - Still respond to user input
 - While reading file in the background

1/23/24

Kubiatowicz CS162 © UCB Spring 2024

Lec 3.33

Multithreaded Programs

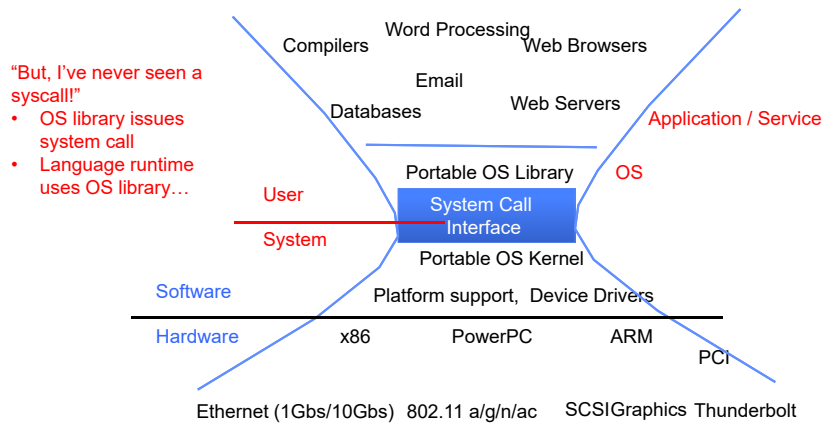
- You know how to compile a C program and run the executable
 - This creates a process that is executing that program
- Initially, this new process has *one thread* in its own address space
 - With code, globals, etc. as specified in the executable
- Q: How can we make a multithreaded process?
- A: Once the process starts, it issues *system calls* to create new threads
 - These new threads are part of the process: they share its address space

1/23/24

Kubiatowicz CS162 © UCB Spring 2024

Lec 3.34

System Calls ("Syscalls")

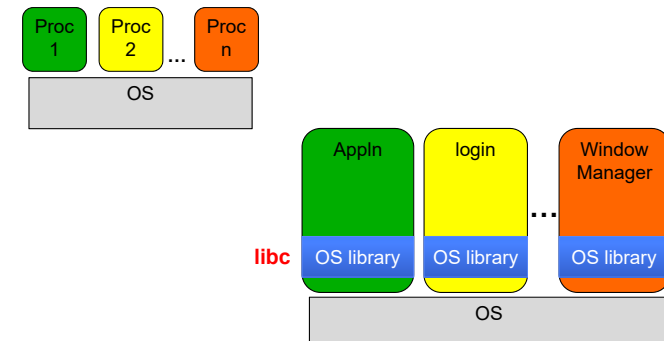


1/23/24

Kubiatowicz CS162 © UCB Spring 2024

Lec 3.35

OS Library Issues Syscalls



1/23/24

Kubiatowicz CS162 © UCB Spring 2024

Lec 3.36

OS Library API for Threads: *pthread*s

Here: the “p” is for “POSIX” which is a part of a standardized API

```
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                  void *(*start_routine)(void*), void *arg);
```

- thread is created executing *start_routine* with *arg* as its sole argument.
- return is implicit call to *pthread_exit*

```
void pthread_exit(void *value_ptr);
```

- terminates the thread and makes *value_ptr* available to any successful join

```
int pthread_join(pthread_t thread, void **value_ptr);
```

- suspends execution of the calling thread until the target *thread* terminates.
- On return with a non-NULL *value_ptr* the value passed to *pthread_exit()* by the terminating thread is made available in the location referenced by *value_ptr*.

prompt% man pthread

<https://pubs.opengroup.org/onlinepubs/7908799/xsh/pthread.h.html>

1/23/24

Kubiatowicz CS162 © UCB Spring 2024

Lec 3.37

Peeking Ahead: System Call Example

- What happens when *pthread_create(...)* is called in a process?

Library:

```
int pthread_create(...) {
    Do some work like a normal fn...
```

```
asm code ... syscall # into %eax
put args into registers %ebx, ...
special trap instruction
```

Kernel:

```
get args from regs
dispatch to system func
Do the work to spawn the new thread
Store return value in %eax
```

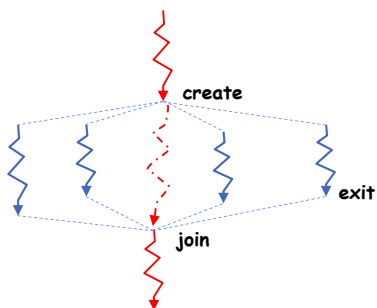
```
get return values from regs
Do some more work like a normal fn...
};
```

1/23/24

Kubiatowicz CS162 © UCB Spring 2024

Lec 3.38

New Idea: Fork-Join Pattern



- Main thread *creates* (forks) collection of sub-threads passing them args to work on...
- ... and then *joins* with them, collecting results.

1/23/24

Kubiatowicz CS162 © UCB Spring 2024

Lec 3.39

pThreads Example

- How many threads are in this program?
- What function does each thread run?
- One possible result:

```
((base) CullerMac19:code04 culler$ ./pthread 4
Main stack: 7ffee2c6b6b8, common: 10cf95048 (162)
Thread #1 stack: 70000d83bef8 common: 10cf95048 (162)
Thread #3 stack: 70000d941ef8 common: 10cf95048 (164)
Thread #2 stack: 70000d8beef8 common: 10cf95048 (165)
Thread #0 stack: 70000d7b8ef8 common: 10cf95048 (163)
```

- Does the main thread join with the threads in the same order that they were created?
 - Yes: Loop calls Join in thread order
- Do the threads exit in the same order they were created?
 - No: Depends on scheduling order!
- Would the result change if run again?
 - Yes: Depends on scheduling order!
- Is this code safe/correct???
- No – threads share are variable that is used without locking and there is a race condition!

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <string.h>

int common = 162;

void *threadfun(void *threadid)
{
    long tid = (long)threadid;
    printf("Thread #%lx stack: %lx common: %lx tid: %ld\n", tid,
          (unsigned long)&tid, (unsigned long)&common, common++);
    pthread_exit(NULL);
}

int main (int argc, char *argv[])
{
    long t;
    int nthreads = 2;
    for (t = 1; t < nthreads; t++)
        nthreads = atoi(argv[1]);

    pthread_t *threads = malloc(nthreads*sizeof(pthread_t));
    printf("Main stack: %lx, common: %lx\n",
          (unsigned long)&tid, (unsigned long)&common, common);
    for (t = 0; t < nthreads; t++)
        if (rc = pthread_create(&threads[t], NULL, threadfun, (void *)t))
            printf("ERROR: return code from pthread_create() is %d\n", rc);
    pthread_exit(NULL);
}

/* last thing in the main thread */
pthread_exit(NULL);
```

1/23/24

Kubiatowicz CS162 © UCB Spring 2024

Lec 3.40

Thread State

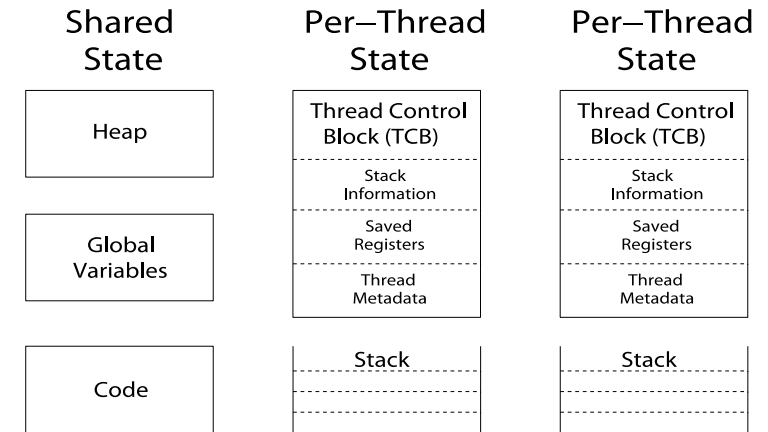
- State shared by all threads in process/address space
 - Content of memory (global variables, heap)
 - I/O state (file descriptors, network connections, etc)
- State “private” to each thread
 - Kept in **TCB = Thread Control Block**
 - CPU registers (including, program counter)
 - Execution stack – what is this?
- Execution Stack
 - Parameters, temporary variables
 - Return PCs are kept while called procedures are executing

1/23/24

Kubiatowicz CS162 © UCB Spring 2024

Lec 3.41

Shared vs. Per-Thread State



1/23/24

Kubiatowicz CS162 © UCB Spring 2024

Lec 3.42

Execution Stack Example

```

A(int tmp) {
A:   if (tmp<2)
A+1:   B();
A+2:   printf(tmp);
      }
      B() {
B:     C();
B+1:   }
      C() {
C:     A(2);
C+1:   }
      A(1);
exit:

```

- Stack holds temporary results
- Permits recursive execution
- Crucial to modern languages

1/23/24

Kubiatowicz CS162 © UCB Spring 2024

Lec 3.43

Execution Stack Example

```

A(int tmp) {
A:   if (tmp<2)
A+1:   B();
A+2:   printf(tmp);
      }
      B() {
B:     C();
B+1:   }
      C() {
C:     A(2);
C+1:   }
      A(1);
exit:

```

Stack
Pointer → **A: tmp=1
ret=exit**

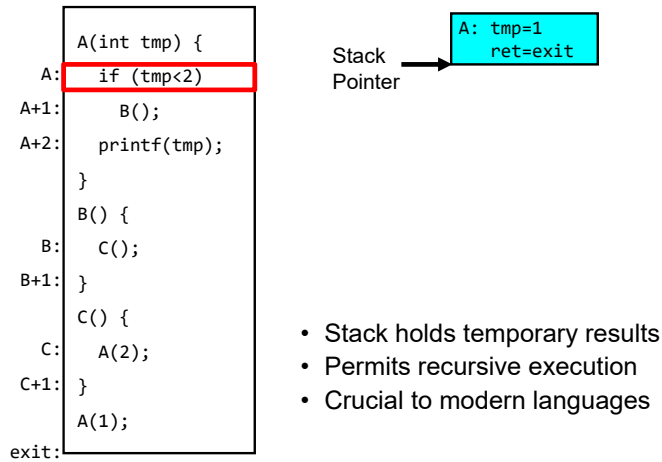
- Stack holds temporary results
- Permits recursive execution
- Crucial to modern languages

1/23/24

Kubiatowicz CS162 © UCB Spring 2024

Lec 3.44

Execution Stack Example

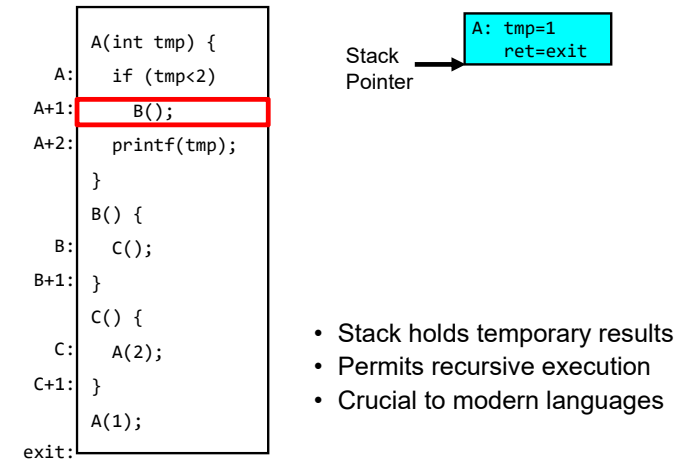


1/23/24

Kubiatowicz CS162 © UCB Spring 2024

Lec 3.45

Execution Stack Example

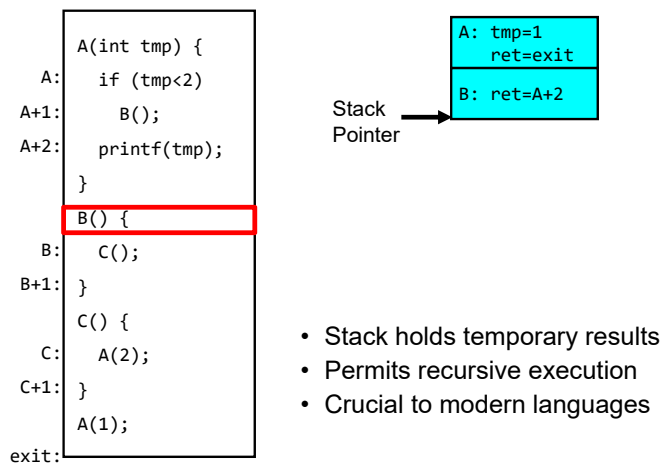


1/23/24

Kubiatowicz CS162 © UCB Spring 2024

Lec 3.46

Execution Stack Example

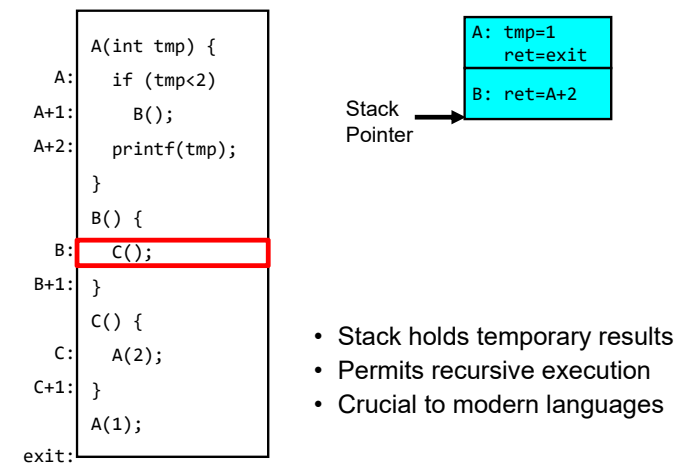


1/23/24

Kubiatowicz CS162 © UCB Spring 2024

Lec 3.47

Execution Stack Example

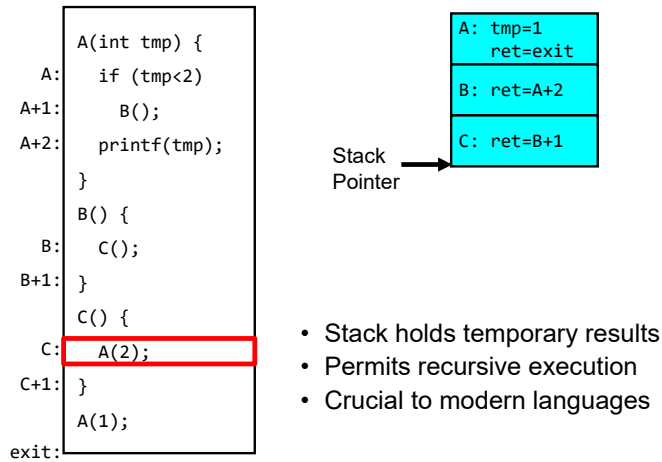


1/23/24

Kubiatowicz CS162 © UCB Spring 2024

Lec 3.48

Execution Stack Example

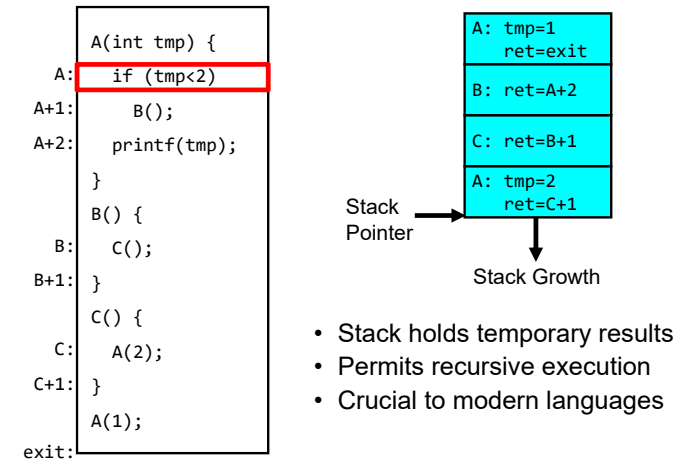


1/23/24

Kubiatowicz CS162 © UCB Spring 2024

Lec 3.49

Execution Stack Example

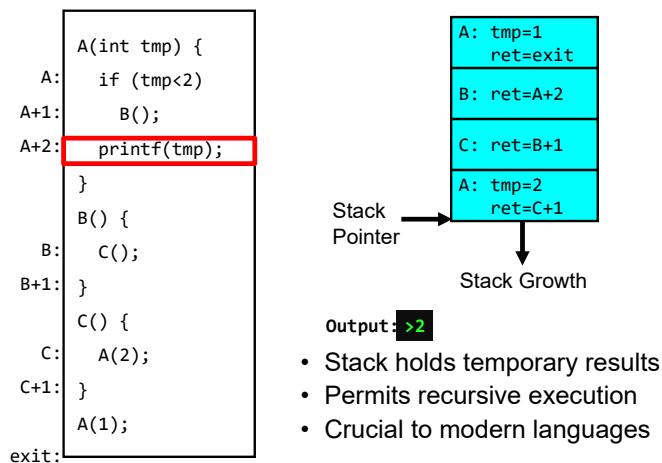


1/23/24

Kubiatowicz CS162 © UCB Spring 2024

Lec 3.50

Execution Stack Example

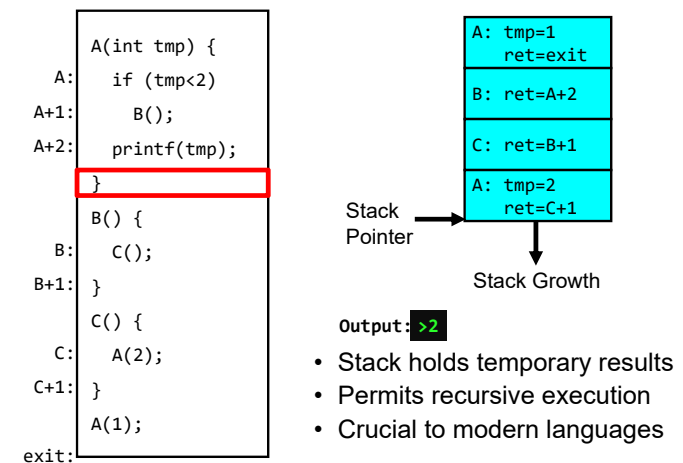


1/23/24

Kubiatowicz CS162 © UCB Spring 2024

Lec 3.51

Execution Stack Example

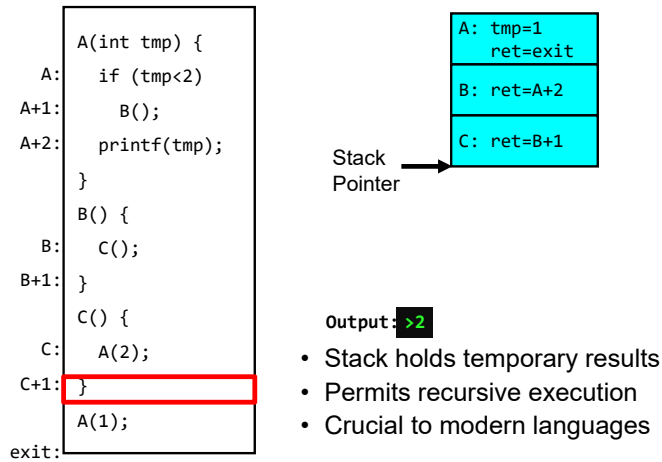


1/23/24

Kubiatowicz CS162 © UCB Spring 2024

Lec 3.52

Execution Stack Example

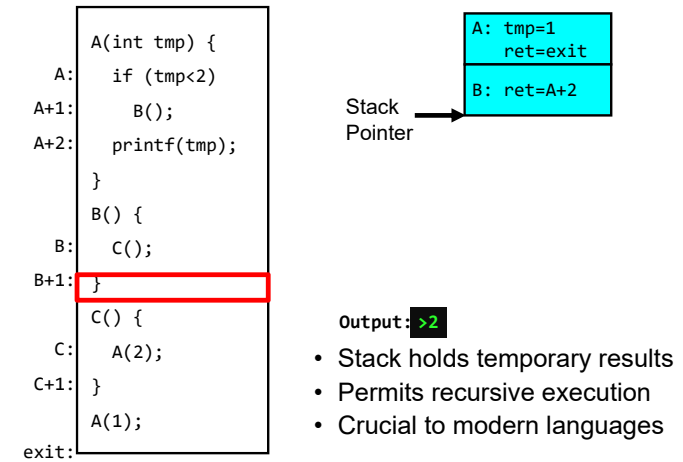


1/23/24

Kubiatowicz CS162 © UCB Spring 2024

Lec 3.53

Execution Stack Example

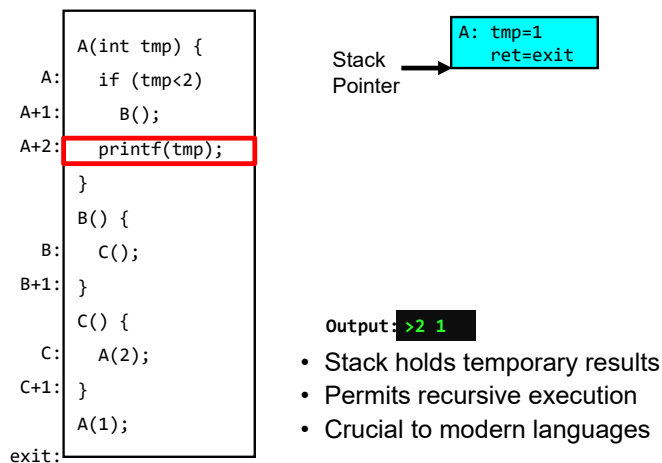


1/23/24

Kubiatowicz CS162 © UCB Spring 2024

Lec 3.54

Execution Stack Example

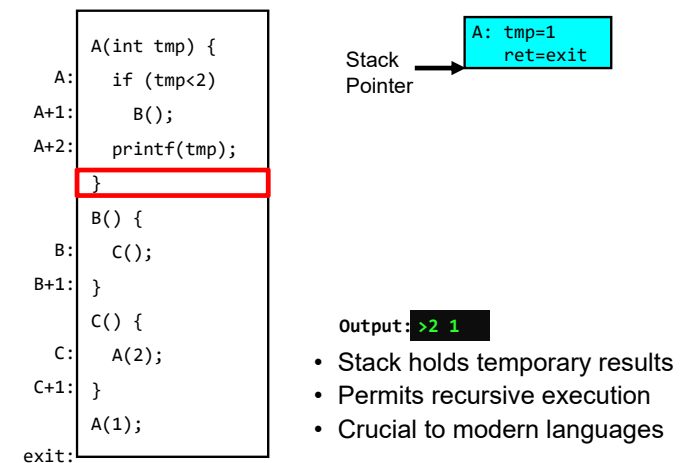


1/23/24

Kubiatowicz CS162 © UCB Spring 2024

Lec 3.55

Execution Stack Example



1/23/24

Kubiatowicz CS162 © UCB Spring 2024

Lec 3.56

Execution Stack Example

```
A(int tmp) {
    if (tmp<2)
        B();
    printf(tmp);
}
B() {
    C();
}
C() {
    A(2);
}
A(1);
```

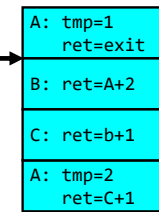
Output: >2 1

- Stack holds temporary results
- Permits recursive execution
- Crucial to modern languages

Execution Stack Example

```
A(int tmp) {
    if (tmp<2)
        B();
    printf(tmp);
}
B() {
    C();
}
C() {
    A(2);
}
A(1);
```

Stack
Pointer

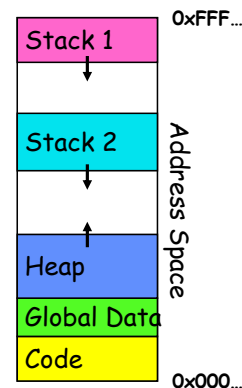


Stack Growth

- Stack holds temporary results
- Permits recursive execution
- Crucial to modern languages

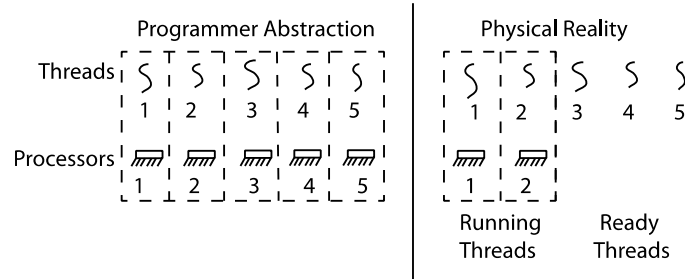
Memory Layout with Two Threads

- Two sets of CPU registers
- Two sets of Stacks
- Issues:
 - How do we position stacks relative to each other?
 - What maximum size should we choose for the stacks?
 - What happens if threads violate this?
 - How might you catch violations?



INTERLEAVING AND NONDETERMINISM
(The beginning of a long discussion!)

Thread Abstraction



- Illusion: Infinite number of processors
- Reality: Threads execute with variable “speed”
 - Programs must be designed to work with any schedule

1/23/24

Kubiatowicz CS162 © UCB Spring 2024

Lec 3.61

Programmer vs. Processor View

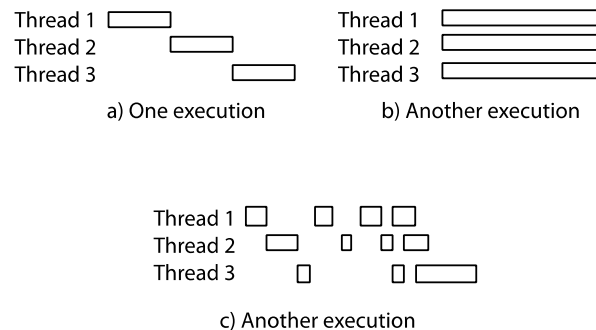
Programmer's View	Possible Execution #1	Possible Execution #2	Possible Execution #3
.	.	.	.
.	.	.	.
.	.	.	.
$x = x + 1;$	$x = x + 1;$	$x = x + 1$	$x = x + 1$
$y = y + x;$	$y = y + x;$	$y = y + x$
$z = x + 5y;$	$z = x + 5y;$	thread is suspended
.	.	other thread(s) run	thread is suspended
.	.	thread is resumed	other thread(s) run
.	thread is resumed
		$y = y + x$
		$z = x + 5y$	$z = x + 5y$

1/23/24

Kubiatowicz CS162 © UCB Spring 2024

Lec 3.62

Possible Executions



1/23/24

Kubiatowicz CS162 © UCB Spring 2024

Lec 3.63

Correctness with Concurrent Threads

- Non-determinism:
 - Scheduler can run threads in **any order**
 - Scheduler can switch threads **at any time**
 - This can make testing very difficult
- *Independent Threads*
 - No state shared with other threads
 - Deterministic, reproducible conditions
- *Cooperating Threads*
 - Shared state between multiple threads
- **Goal: Correctness by Design**

1/23/24

Kubiatowicz CS162 © UCB Spring 2024

Lec 3.64

Race Conditions: Example 1

- Initially $x == 0$ and $y == 0$

<u>Thread A</u>	<u>Thread B</u>
$x = 1;$	$y = 2;$

- What are the possible values of x below after all threads finish?
- Must be 1. Thread B does not interfere

1/23/24

Kubiatowicz CS162 © UCB Spring 2024

Lec 3.65

Race Conditions: Example 2

- Initially $x == 0$ and $y == 0$

<u>Thread A</u>	<u>Thread B</u>
$x = y + 1;$	$y = 2;$
	$y = y * 2;$

- What are the possible values of x below?
- 1 or 3 or 5 (non-deterministically)
- Race Condition: Thread A races against Thread B!**

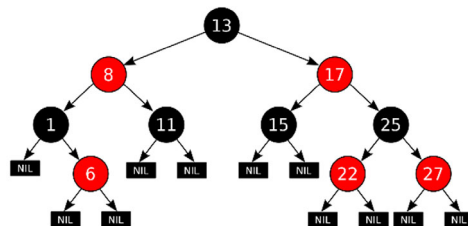
1/23/24

Kubiatowicz CS162 © UCB Spring 2024

Lec 3.66

Example: Shared Data Structure

Thread A
Insert(3)



Thread B
Insert(4)
Get(6)

Tree-Based Set Data Structure

1/23/24

Kubiatowicz CS162 © UCB Spring 2024

Lec 3.67

Relevant Definitions

- Synchronization: Coordination among threads, usually regarding shared data
- Mutual Exclusion:** Ensuring only one thread does a particular thing at a time (one thread *excludes* the others)
 - Type of synchronization
- Critical Section:** Code exactly one thread can execute at once
 - Result of mutual exclusion
- Lock:** An object only one thread can hold at a time
 - Provides mutual exclusion

1/23/24

Kubiatowicz CS162 © UCB Spring 2024

Lec 3.68

Locks

- Locks provide two **atomic** operations:
 - **Lock.acquire()** – wait until lock is free; then mark it as busy
 - » After this returns, we say the calling thread *holds* the lock
 - **Lock.release()** – mark lock as free
 - » Should only be called by a thread that currently holds the lock
 - » After this returns, the calling thread no longer holds the lock
- For now, don't worry about how to implement locks!
 - We'll cover that in substantial depth later on in the class

1/23/24

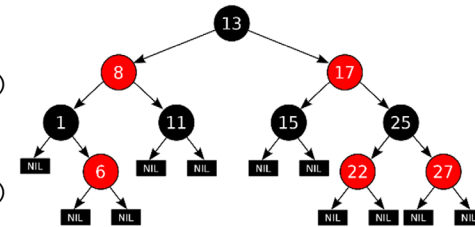
Kubiatowicz CS162 © UCB Spring 2024

Lec 3.69

Thread A

Insert(3)

- Lock.acquire()
- Insert 3 into the data structure
- Lock.release()



Tree-Based Set Data Structure

Thread B

Insert(4)

- Lock.acquire()
 - Insert 4 into the data structure
 - Lock.release()
- Get(6)
- Lock.acquire()
 - Check for membership
 - Lock.release()

1/23/24

Kubiatowicz CS162 © UCB Spring 2024

Lec 3.70

OS Library Locks: *pthread*s

```
int pthread_mutex_init(pthread_mutex_t *mutex,
                       const pthread_mutexattr_t *attr)
```

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

You'll get a chance to use these in Homework 1

1/23/24

Kubiatowicz CS162 © UCB Spring 2024

Lec 3.71

Our Example

Critical section

```
int common = 162;
pthread_mutex_t common_lock = PTHREAD_MUTEX_INITIALIZER;

void *threadfun(void *threadid)
{
    long tid = (long)threadid;
    pthread_mutex_lock(&common_lock);
    int my_common = common++;
    pthread_mutex_unlock(&common_lock);

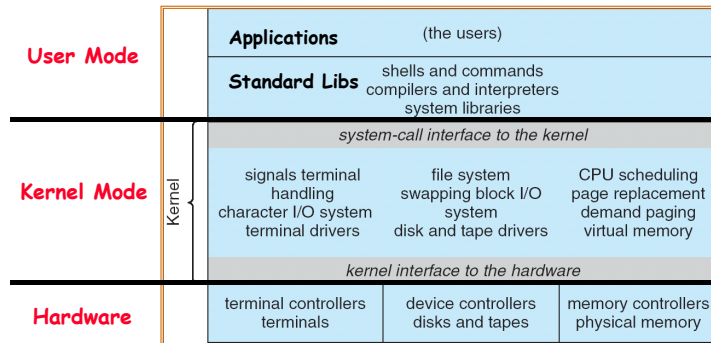
    printf("Thread #%lx stack: %lx common: %lx (%d)\n", tid,
           (unsigned long) &tid,
           (unsigned long) &common, my_common);
    pthread_exit(NULL);
}
```

1/23/24

Kubiatowicz CS162 © UCB Spring 2024

Lec 3.72

Recall: UNIX System Structure

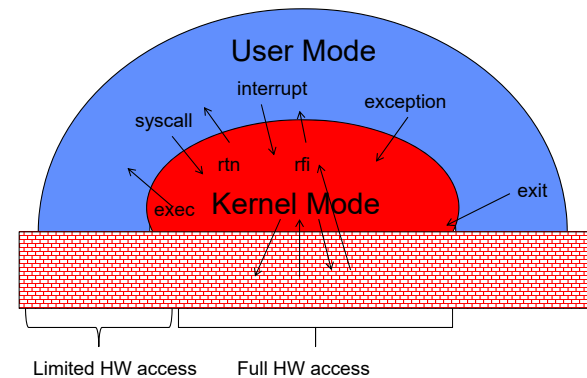


1/23/24

Kubiatowicz CS162 © UCB Spring 2024

Lec 3.73

Recall: Life of a Process



1/23/24

Kubiatowicz CS162 © UCB Spring 2024

Lec 3.74

3 types of Kernel Mode Transfer

- **Syscall**
 - Process requests a system service, e.g., exit
 - Like a function call, but “outside” the process
 - Does not have the address of the system function to call
 - Like a Remote Procedure Call (RPC) – for later
 - Marshall the syscall id and args in registers and exec syscall
- **Interrupt**
 - External asynchronous event triggers context switch
 - eg. Timer, I/O device
 - Independent of user process
- **Trap or Exception**
 - Internal synchronous event in process triggers context switch
 - e.g., Protection violation (segmentation fault), Divide by zero, ...

1/23/24

Kubiatowicz CS162 © UCB Spring 2024

Lec 3.75

Implementing Safe Kernel Mode Transfers

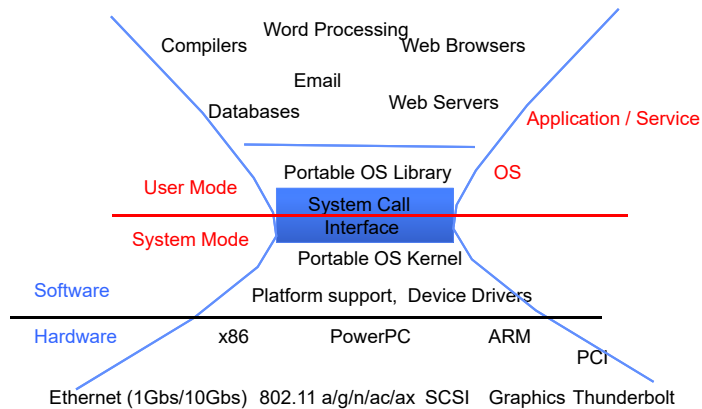
- **Important aspects:**
 - Controlled transfer into kernel (e.g., syscall table)
 - **Separate kernel stack!**
- **Carefully constructed kernel code packs up the user process state and sets it aside**
 - Details depend on the machine architecture
 - More on this next time
- **Should be impossible for buggy or malicious user program to cause the kernel to corrupt itself!**

1/23/24

Kubiatowicz CS162 © UCB Spring 2024

Lec 3.76

Recall: System Call Interface: A Narrow Waist



1/23/24

Kubiawicz CS162 © UCB Spring 2024

Lec 3.77

Kernel System Call Handler

- Vector through well-defined syscall entry points!
 - Table mapping *system call number* to *handler*
- Locate arguments
 - In registers or on user (!) stack
- Copy arguments
 - From user memory into kernel memory
 - Protect kernel from malicious code evading checks
- Validate arguments
 - Protect kernel from errors in user code
- Copy results back
 - Into user memory

1/23/24

Kubiawicz CS162 © UCB Spring 2024

Lec 3.78

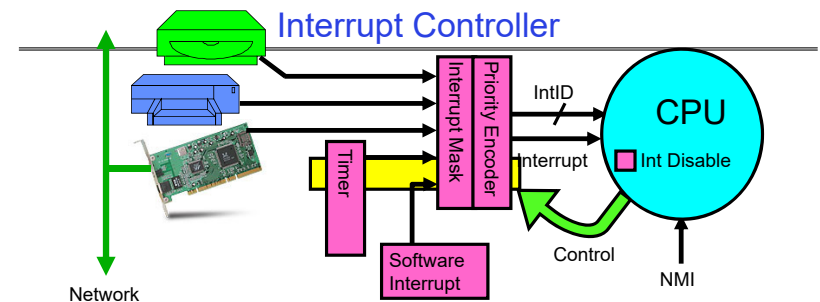
Hardware support: Interrupt Control

- Interrupt processing not visible to the user process:
 - Occurs between instructions, restarted transparently
 - No change to process state
 - What can be observed even with perfect interrupt processing?
- Interrupt Handler invoked with interrupts 'disabled'
 - Re-enabled upon completion
 - Non-blocking (run to completion, no waits)
 - Pack up in a queue and pass off to an OS thread for hard work
 - » wake up an existing OS thread

1/23/24

Kubiawicz CS162 © UCB Spring 2024

Lec 3.79



- Interrupts invoked with interrupt lines from devices
- Interrupt controller chooses interrupt request to honor
 - Interrupt identity specified with ID line
 - Mask enables/disables interrupts
 - Priority encoder picks highest enabled interrupt
 - Software Interrupt Set/Cleared by Software
- CPU can disable all interrupts with internal flag
- Non-Maskable Interrupt line (NMI) can't be disabled

1/23/24

Kubiawicz CS162 © UCB Spring 2024

Lec 3.80

How do we take interrupts safely?

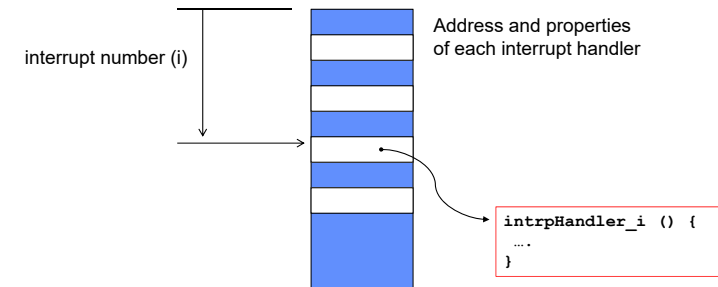
- **Interrupt vector**
 - Limited number of entry points into kernel
- Kernel interrupt stack
 - Handler works regardless of state of user code
- Interrupt masking
 - Handler is non-blocking
- Atomic transfer of control
 - “Single instruction”-like to change:
 - » Program counter
 - » Stack pointer
 - » Memory protection
 - » Kernel/user mode
- Transparent restartable execution
 - User program does not know interrupt occurred

1/23/24

Kubiatowicz CS162 © UCB Spring 2024

Lec 3.81

Interrupt Vector



- Where else do you see this dispatch pattern?
 - System Call
 - Exceptions

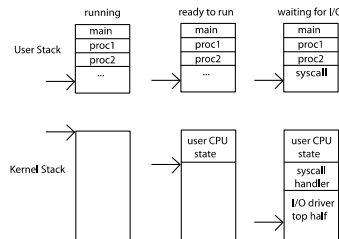
1/23/24

Kubiatowicz CS162 © UCB Spring 2024

Lec 3.82

Need for Separate Kernel Stacks

- Kernel needs space to work
- Cannot put anything on the user stack (Why?)
- Two-stack model
 - OS thread has interrupt stack (located in kernel memory) plus User stack (located in user memory)
 - Syscall handler copies user args to kernel space before invoking specific function (e.g., open)
 - Interrupts (???)

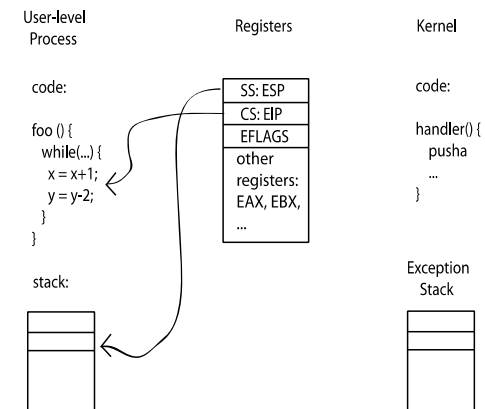


1/23/24

Kubiatowicz CS162 © UCB Spring 2024

Lec 3.83

Before

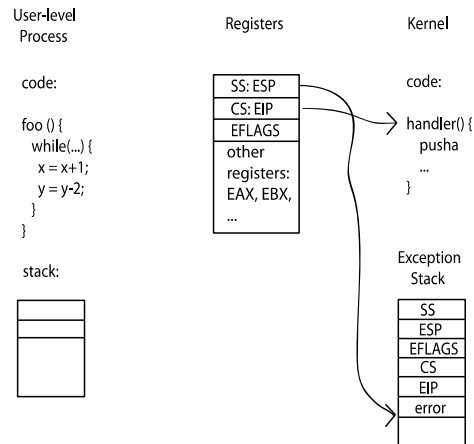


1/23/24

Kubiatowicz CS162 © UCB Spring 2024

Lec 3.84

During Interrupt/System Call



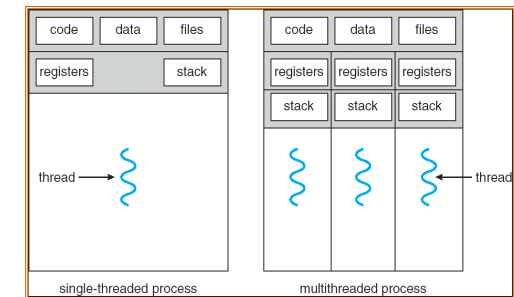
1/23/24

Kubiatowicz CS162 © UCB Spring 2024

Lec 3.85

Managing Processes

- How to manage process state?
 - How to create a process?
 - How to exit from a process?
- Remember: Everything outside of the kernel is running in a process!
 - Including the shell! (Homework 2)
- Processes are created and managed... by processes!



1/23/24

Kubiatowicz CS162 © UCB Spring 2024

Lec 3.86

Bootstrapping

- If processes are created by other processes, how does the first process start?
- First process is started by the kernel
 - Often configured as an argument to the kernel *before* the kernel boots
 - Often called the “init” process
- After this, all processes on the system are created by other processes

1/23/24

Kubiatowicz CS162 © UCB Spring 2024

Lec 3.87

Next time: Process Management API

- `exit` – terminate a process
- `fork` – copy the current process
- `exec` – change the *program* being run by the current process
- `wait` – wait for a process to finish
- `kill` – send a *signal* (interrupt-like notification) to another process
- `sigaction` – set handlers for signals

1/23/24

Kubiatowicz CS162 © UCB Spring 2024

Lec 3.88

Conclusion

- Threads are the OS unit of concurrency
 - Abstraction of a virtual CPU core
 - Can use `pthread_create`, etc., to manage threads within a process
 - They share data → need synchronization to avoid data races
- Processes consist of one or more threads in an address space
 - Abstraction of the machine: execution environment for a program
 - Can use `fork`, `exec`, etc. to manage threads within a process
- We saw the role of the OS library
 - Provide API to programs
 - Interface with the OS to request services