Processes and Threads

CSCI 237: Computer Organization
34th Lecture, Monday, December 1, 2025

Kelly Shaw

Administrative Details

- Lab #6 due Friday at 5pm
- Read CSAPP 12.1—12.4
- Review session
  - Thursday or Friday of next week?
- Colloquium tomorrow at 2:35pm
  - Olivia Weng, UCSD
  - Codesigning Hardware and Software for Efficient AI

Last Time

- Dynamic Memory Allocation (Ch 9.9)
  - Tracking Free Blocks
    - Explicit Lists
    - Segregated Lists

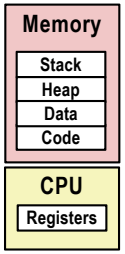
Today: Processes and Threads

- Processes
- Threads

Processes

- Definition: A *process* is an instance of a running program.
  - One of the most profound ideas in computer science
  - Not the same as "program" or "processor"

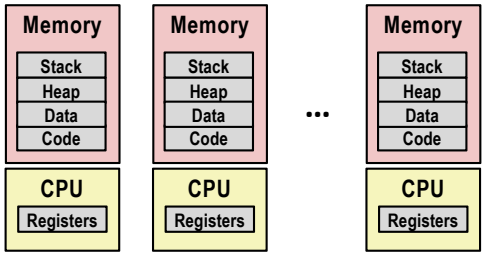
- Process provides each program with two key abstractions:
  - ***Logical control flow***
    - Each program seems to have exclusive use of the CPU
    - Provided by kernel mechanism called *context switching*
  - ***Private address space***
    - Each program seems to have exclusive use of main memory.
    - Provided by kernel mechanism called *virtual memory*



5

5

Multiprocessing: The Illusion

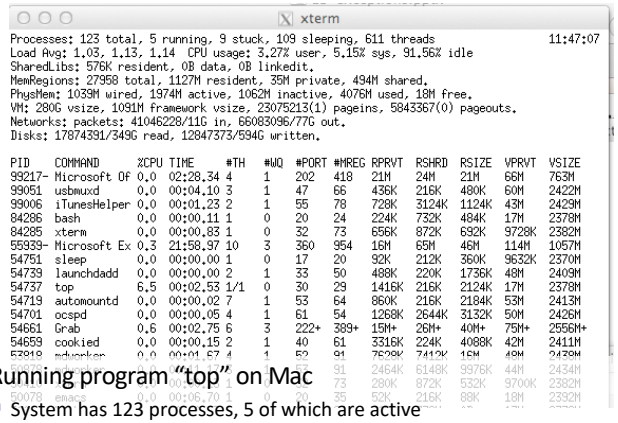


- Computer runs many processes simultaneously
  - Applications for one or more users
    - Web browsers, email clients, editors, …
  - Background tasks
    - Monitoring network & I/O devices

6

6

Multiprocessing Example

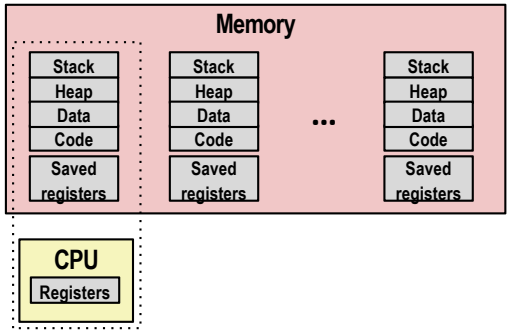


- Running program "top" on Mac
  - System has 123 processes, 5 of which are active
  - Identified by Process ID (PID)

7

7

Multiprocessing: The (Traditional) Reality

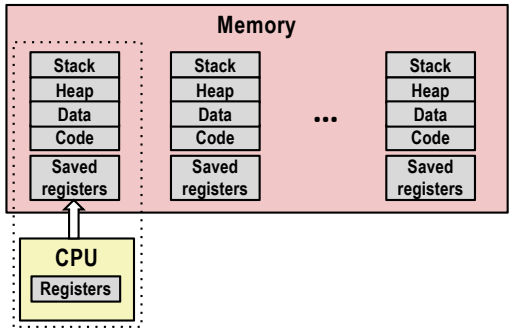


- Single processor executes multiple processes concurrently
  - Process executions interleaved (multitasking)
  - Address spaces managed by virtual memory system
  - Register values for non-executing processes saved in memory

8

8

Multiprocessing: The (Traditional) Reality

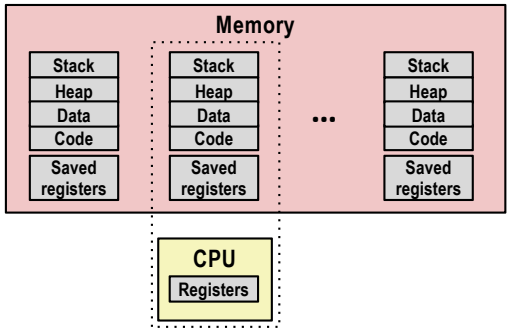


- Save current registers in memory

9

9

Multiprocessing: The (Traditional) Reality

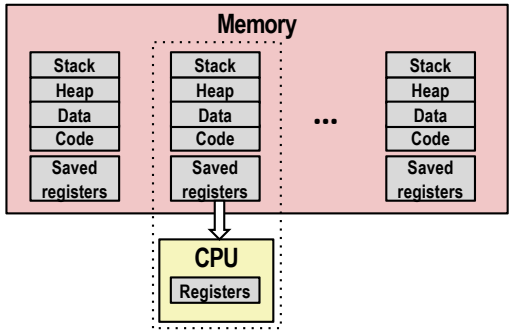


- Schedule next process for execution

10

10

Multiprocessing: The (Traditional) Reality

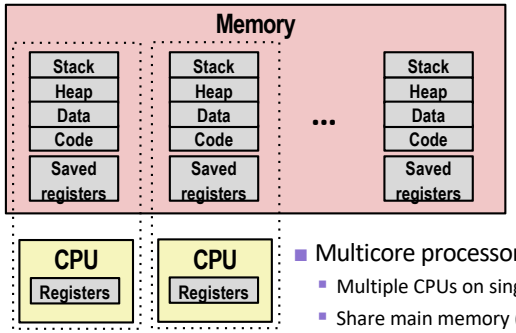


- Load saved registers and switch address space (context switch)

11

11

Multiprocessing: The (Modern) Reality



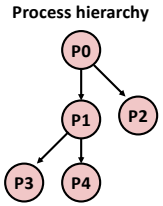
- Multicore processors
  - Multiple CPUs on single chip
  - Share main memory (and some of the caches)
  - Each can execute a separate process
    - Scheduling of processors onto cores done by kernel (OS)

12

12

Processes Can Create Other Processes

- A *process* can create another process.
  - The created process is a child process of its parent.
- The child process can
  - continue to execute the original executable that created it (with a copy of the parent's resources) or
  - start running a new executable
- The parent and child process do not share anything, but the parent process will be notified when the child process completes.
- A parent process can wait for a child process to complete before it continues its execution.

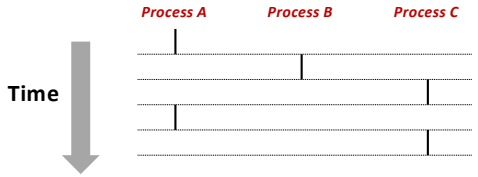


13

13

Concurrent Processes

- Each process is a logical control flow.
- Two processes *run concurrently* (*are concurrent)* if their flows overlap in time
- Otherwise, they are *sequential*
- Examples (running on single core):
  - Concurrent: A & B, A & C
  - Sequential: B & C

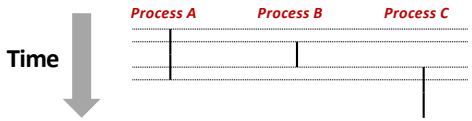


14

14

User View of Concurrent Processes

- Control flows for concurrent processes are physically disjoint in time
- However, we can think of concurrent processes as running in parallel with each other

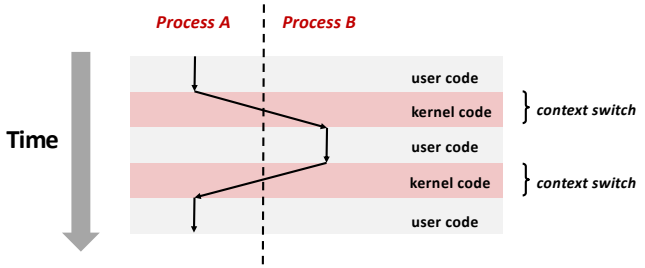


15

15

Context Switching

- Processes are managed by a shared chunk of memory-resident OS code called the *kernel*
  - Important: the kernel is not a separate process, but rather runs as part of some existing process.
- Control flow passes from one process to another via a *context switch*

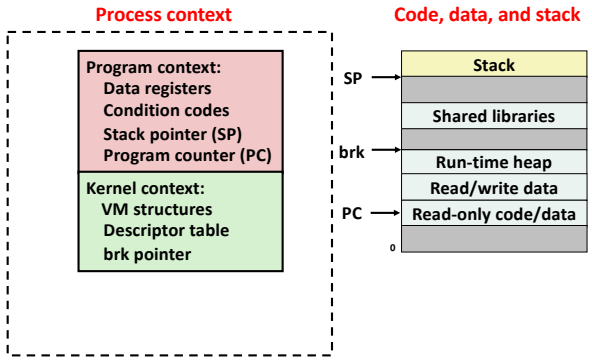


16

16

Traditional View of a Process

- Process = process context + code, data, and stack

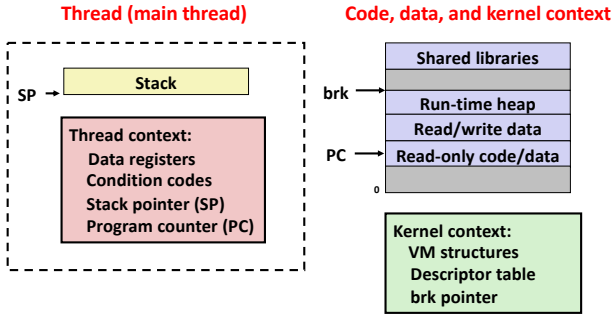


17

17

Alternate View of a Process

- Process = thread + code, data, and kernel context

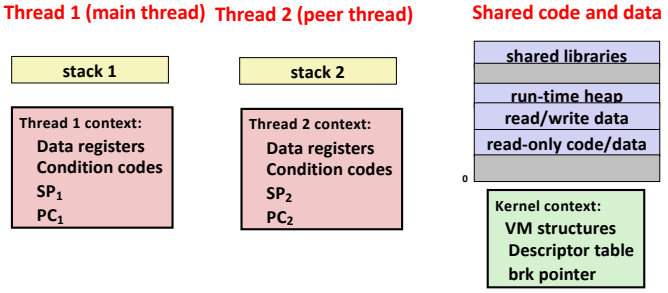


18

18

A Process With Multiple Threads

- Multiple threads can be associated with a process
  - Each thread has its own logical control flow
  - Each thread shares the same code, data, and kernel context
  - Each thread has its own stack for local variables
    - but **not** protected from other threads
  - Each thread has its own thread id (TID)

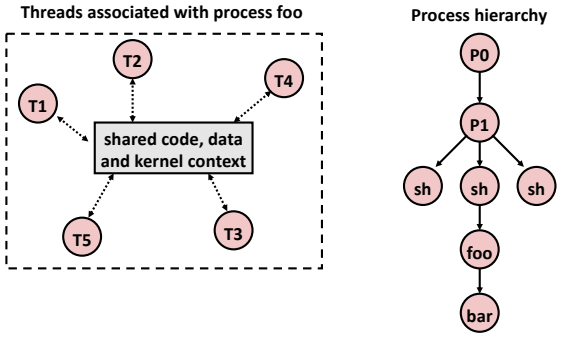


19

19

Logical View of Threads

- Threads associated with process form a pool of "peers"
  - Unlike processes which form a tree hierarchy



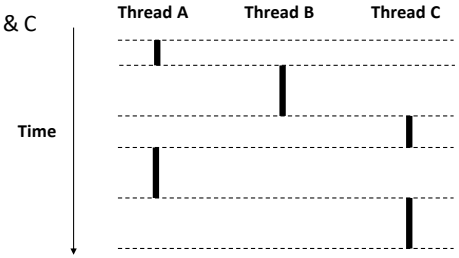
20

20

Concurrent Threads

- Two threads are *concurrent* if their flows "overlap" in time
- Otherwise, they are sequential w.r.t. each other

- Examples:
  - Concurrent: A & B, A & C
  - Sequential: B & C

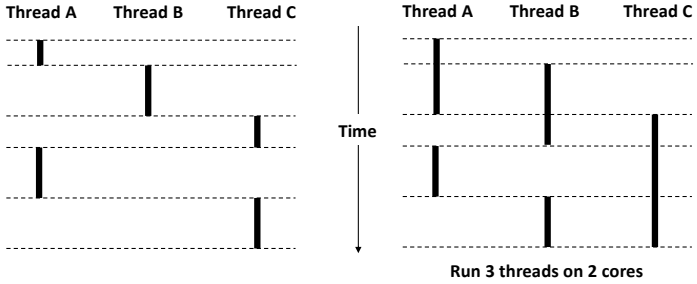


21

21

Concurrent Thread Execution

- Single Core Processor
  - Simulate parallelism by time slicing
- Multi-Core Processor
  - Can have true parallelism



22

22

Threads vs. Processes

- How threads and processes are similar
  - Each has its own logical control flow
  - Each can run concurrently with others (possibly on different cores)
  - Each is *context switched*
- How threads and processes are different
  - Threads share all code and data (except local stacks)
    - Processes do not
  - Threads are somewhat less expensive than processes
    - Process control (creating and reaping) twice as expensive as thread control
    - Linux numbers:
      - ~20K cycles to create and reap a process
      - ~10K cycles (or less) to create and reap a thread

23

23

Posix Threads (Pthreads) Interface

- *Pthreads:* Standard interface for ~60 functions that manipulate threads from C programs
  - Creating and reaping threads
    - `pthread_create()`
    - `pthread_join()`
  - Determining your thread ID
    - `pthread_self()`
  - Terminating threads
    - `pthread_cancel()`
    - `pthread_exit()`
    - `exit()` [terminates all threads]
    - `return` [terminates current thread]
  - Synchronizing access to shared variables
    - `pthread_mutex_init`
    - `pthread_mutex_[un]lock`

24

24

The Pthreads "hello, world" Program

```
/*
 * hello.c - Pthreads "hello, world" program
 */
#include "csapp.h"
void *thread(void *vargp);

int main(int argc, char** argv)
{
    pthread_t tid;
    pthread_create(&tid, NULL, thread, NULL);
    pthread_join(tid, NULL);
    return 0;
}
```

hello.c

Thread ID

Thread attributes (usually NULL)

Thread routine

*Thread arguments (void *p)*

*Return value (void **p)*

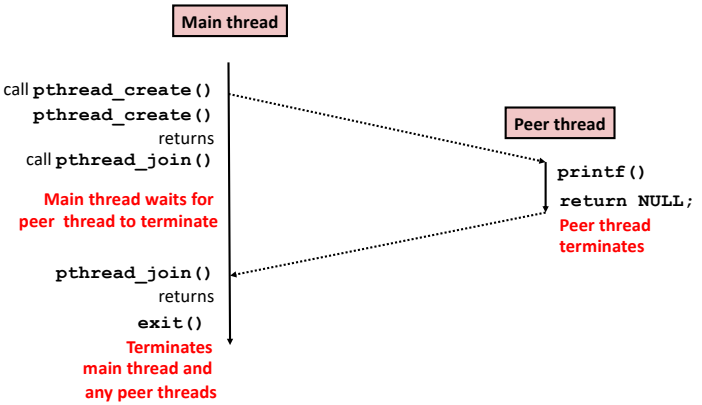
```
void *thread(void *vargp) /* thread routine */
{
    printf("Hello, world!\n");
    return NULL;
}
```

hello.c

25

25

Execution of Threaded "hello, world"



26

26

Basic Pthreads program setup

- `#include <pthread.h>`
- When compiling, link in pthreads library

```
gcc -g -o hello hello.c -lpthread
```

- When running, just run as normal

```
./hello
```

27

27

Issues with Threaded Programs

- When threads concurrently read/write shared memory, program behavior is undefined (called a data race)
  - Two threads write to the same variable; which one should win?
- Thread schedule is non-deterministic
  - Behavior changes when re-run program
- Compiler/hardware instruction reordering
- Multi-word operations are not atomic
- All functions called by a thread must be *thread-safe*

28

28

Question: What is the outcome of this code?

x = 0

Thread 1	Thread 2
`x=x+1;`	`x=x+1;`
`print x;`	`print x;`

29

29

Terminology

- Race condition: output of a concurrent program depends on the order of operations between threads
- Mutual exclusion: only one thread does a particular thing at a time
  - Critical section: piece of code that only one thread can execute at once
- Synchronization primitive: construct used to coordinate use of shared data in threaded programs
- Lock: prevent someone from doing something
  - Lock before entering critical section, before accessing shared data
  - Unlock when leaving, after done accessing shared data
  - Wait if locked (all synchronization involves waiting!)

30

30

```
int num_threads;
int val;

void *Hello(void *rank)
{
  int tmp = val+1;
  val = tmp;
  return NULL;
}
```

```
int main(int argc, char *argv[])
{
  long pthread;
  num_threads = 4;
  val = 0;
  pthread_t ids[num_threads];

  for(long i = 0; i < num_threads; i++){
    pthread_create(&ids[i], NULL, Hello,
                   (void*)i);
  }
  for(int i = 0; i < num_threads; i++){
    pthread_join(ids[i], NULL);
  }
  printf("Value of val %d\n", val);
  return 0;
}
```

31

31

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
int num_threads;
int val;

void *Hello(void *rank)
{
  pthread_mutex_lock(&mutex);
  int tmp = val+1;
  val = tmp;
  pthread_mutex_unlock(&mutex);

  return NULL;
}
```

```
int main(int argc, char *argv[])
{
  long pthread;
  num_threads = 4;
  val = 0;
  pthread_t ids[num_threads];
  pthread_mutex_init(&mutex, NULL);

  for(long i = 0; i < num_threads; i++){
    pthread_create(&ids[i], NULL,
                   Hello, (void*)i);
  }
  for(int i = 0; i < num_threads; i++){
    pthread_join(ids[i], NULL);
  }
  printf("Value of val %d\n", val);
  return 0;
}
```

32

32

Pros and Cons of Threaded Programs

- + Easy to share data structures between threads
  - e.g., logging information, file cache
- + Threads are more efficient than processes
  - Context switches are smaller, threads are more lightweight
- – Unintentional sharing can introduce subtle and hard-to-reproduce errors!
  - The ease with which data can be shared is both the greatest strength and the greatest weakness of threads
  - Hard to know which data shared & which private
  - Hard to detect by testing
    - Probability of bad race outcome very low
    - But nonzero!

33