EECS-343 Operating Systems
Lecture 10:
Threads

Steve Tarzia

Spring 2019

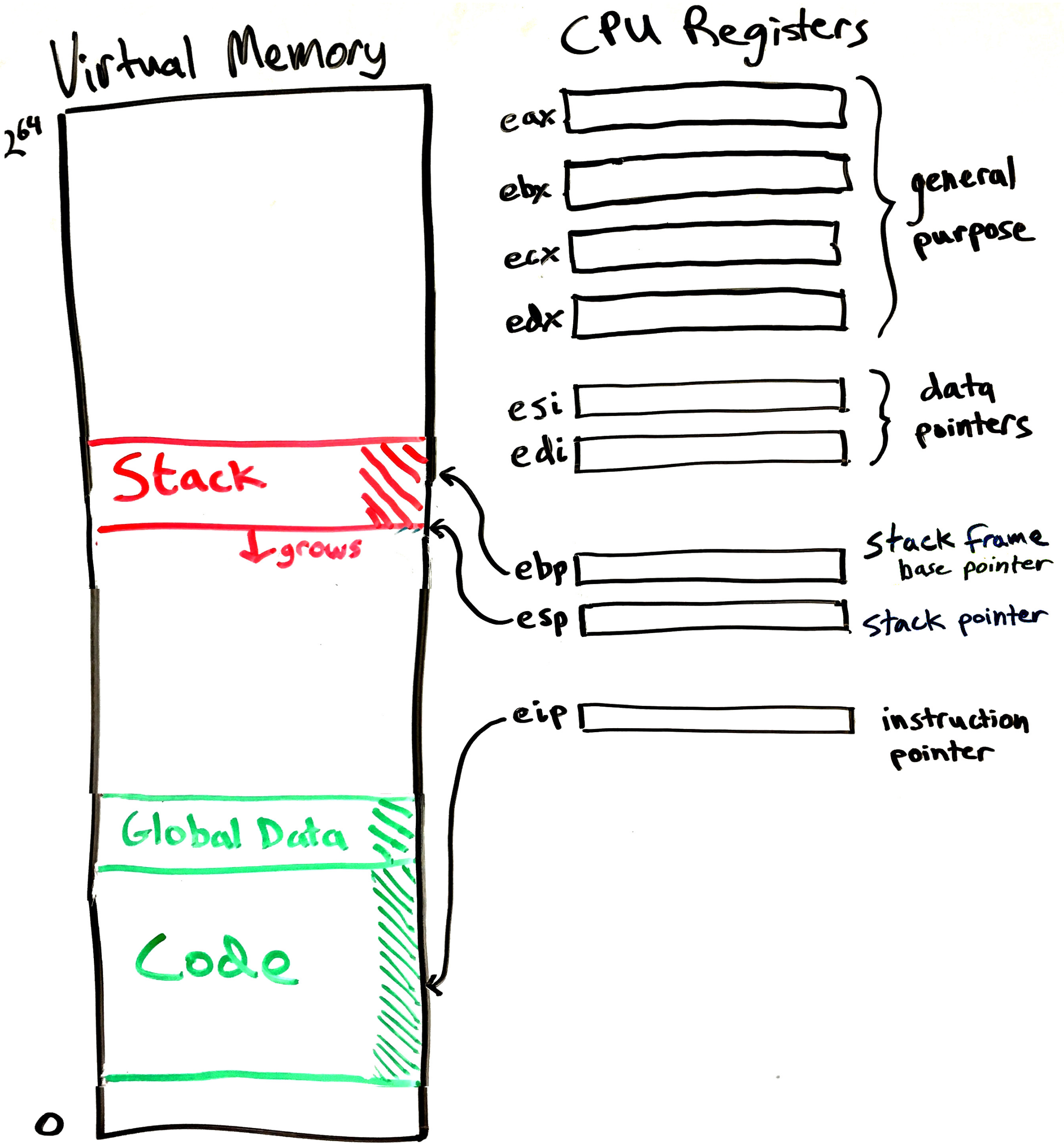
Northwestern

Announcements

- Midterm exams are being graded as we speak.
- Project 2 was due yesterday
- Project 3 is due in two weeks from yesterday
    - Must implement threads in xv6
    - Should be easier than the last assignment
    - There is an extra credit section
- Homework 2 will be out soon.
- Drop deadline is on Friday.
    - I can tell you how you're doing if you're considering dropping.

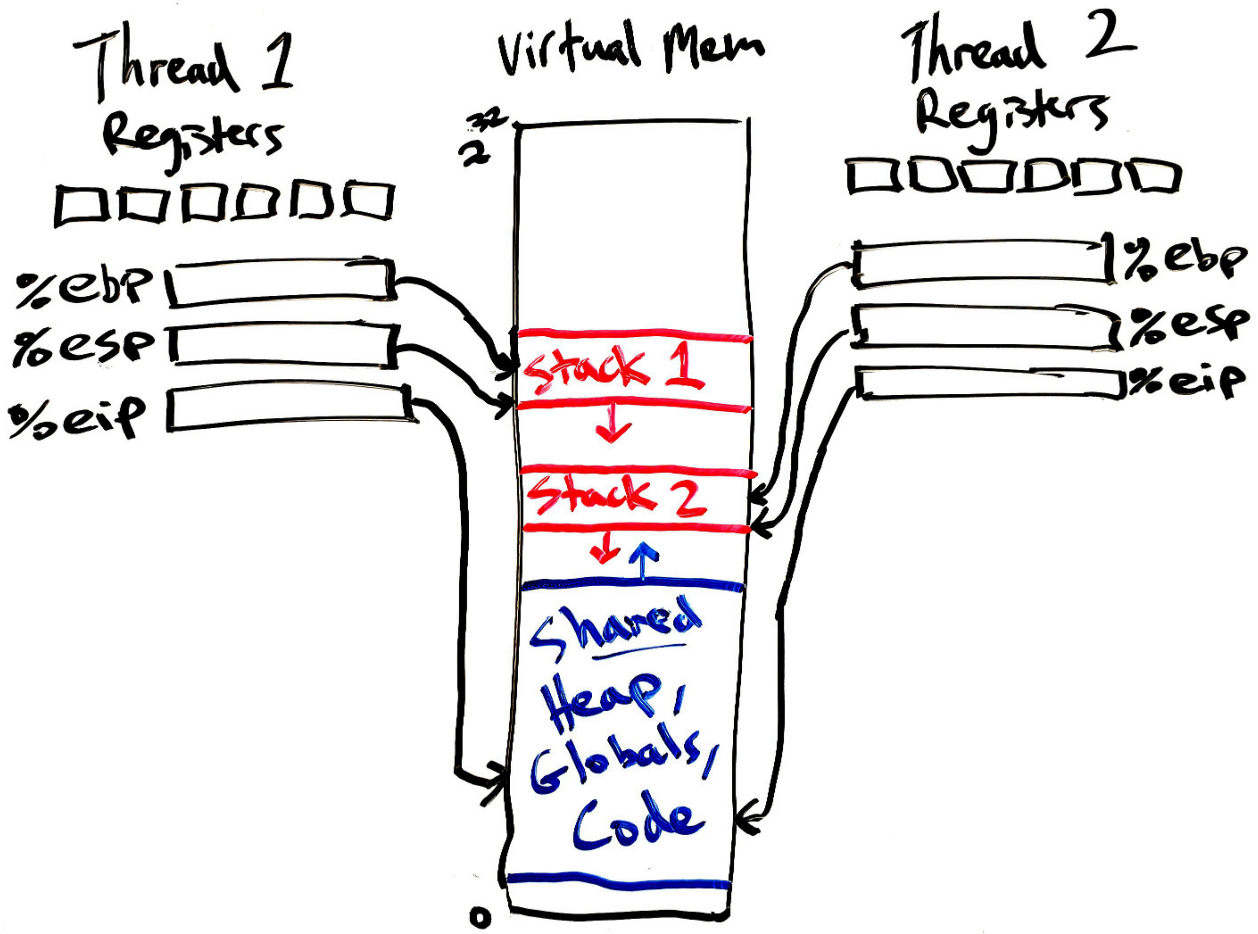
Threads

- So far, we have discussed ***single-threaded*** processes.
- A thread is a part of a process indicating ***where*** it's executing.
- OS scheduler actually schedules threads, not processes.



Multi-threaded processes

- A ***multi-threaded*** process can execute in ***parallel***.
- A thread is like a process, but it **shares all its virtual memory** with all other threads of the same process.
- Each thread has:
  - Its own set of ***register values***
    (Including the instruction pointer)
  - Its own ***stack***



Why use threads?

- Allows a process to work in ***parallel*** and use multiple CPU cores to get work done faster.
- Allows slow tasks to be done in a ***background*** thread.
  - For example:
    - Fetch an image for a website (I/O bound)
    - Save a document to disk (I/O bound)
    - Transcode a media file (CPU bound)
  - This is useful even on machines with a single CPU core.
  - For GUI applications, allows main UI thread to be responsive.
    - UI thread will use little CPU time and retain high priority in MLFQ.
    - Disk/network I/O will not block the UI thread.
- ***Shared memory*** allows the threads to easily coordinate.
  - For example, results can be stored in a global data structure.

Why do we need a stack for each thread?

- Remember that the stack stores:
  - Local function variables
  - Function parameters
  - Return addresses
- CPU needs a stack to track it's progress through C-style functions.
- Each thread takes its own path through the code.
  - So, every thread needs its own stack.
- A thread ***usually*** should not access another thread's stack
  - The stack is ***thread local*** storage.

Thread creation example

```
#include <stdio.h>
#include <assert.h>
#include <pthread.h>

void *mythread(void *arg) {
    printf("%s\n", (char *) arg);
    return NULL;
}

int
main(int argc, char *argv[]) {
    pthread_t p1, p2;
    br int rc;
    printf("main: begin\n");
    rc = pthread_create(&p1, NULL, mythread, "A"); assert(rc == 0);
    rc = pthread_create(&p2, NULL, mythread, "B"); assert(rc == 0);
    // join waits for the threads to finish
    rc = pthread_join(p1, NULL); assert(rc == 0);
    rc = pthread_join(p2, NULL); assert(rc == 0);
    printf("main: end\n");
    return 0;
}
```

- “pthread” refers to the standard **POSIX thread** interface.
  - POSIX is a set of standards for Unix-style OSes.

Thread creation vs. Fork

Thread creation

- `pthread_create()`
  - Creates a thread
  - *Shares* all memory with all threads of the process.
  - Scheduled independently of parent
- `pthread_join()`
  - Waits for a particular thread to finish
- Can communicate by reading/writing (shared) global variables.

Forking a process

- `fork()`
  - Creates a single-threaded process
  - *Copies* all memory from parent
    - Can be quick using copy-on-write
  - Scheduled independently of parent
- `waitpid()`
  - Waits for a particular child process to finish
- Can communicate by setting up shared memory, pipes, reading/writing files, or using sockets (network).

Concurrency can create tricky problems

```c
#include <stdio.h>
#include <pthread.h>

static volatile int counter = 0;
static const int LOOPS = 1e7;

void* mythread(void* arg) {
  printf("%s: begin\n", (char*)arg);
  int i;
  for (i=0; i<LOOPS; i++) {
    counter++;
  }
  printf("%s: done\n", (char*)arg);
  return NULL;
}
```

```c
int main(int argc, char* argv[]) {
  pthread_t p1, p2;
  printf("main: begin (counter = %d)\n", counter);
  pthread_create(&p1, NULL, mythread, "A");
  pthread_create(&p2, NULL, mythread, "B");

  // wait for threads to finish
  pthread_join(p1, NULL);
  pthread_join(p2, NULL);
  printf("main: done with both (counter = %d, goal 
was %d)\n",
      counter, 2*LOOPS);
}
```

- Start two threads, each of which increments a shared global **counter** variable 10^7 times.
- The **`volatile`** keyword tells the compiler that the counter variable may change unexpectedly (in this case, changed by the other thread).

Test parallel_count1.c

- https://gist.github.com/starzia/b6456d74be2f3ab12a0dd4cbff252717
- … or download it from Canvas.
- Compile with “gcc -lpthread parallel_count1.c”

What's the problem?

- We have seen with the fork syscall that the scheduler is unpredictable
  - We don't know which of the two threads will run first and for how long.
  - But is this a problem?
  - Why does it matter who increments the counter first?
  - The net result should be 20,000,000 regardless, right?
  - Actually, there is a serious bug
  - It will yield a ***different result every time!***

```
$ time ./a.out
main: begin (counter = 0)
A: begin
B: begin
A: done
B: done
main: done with both
(counter = 10416197, goal was 20000000)
```

- You have to understand the low-level behavior to find the problem.
  - In short, the "counter++" operation is not ***atomic***.

Incrementing a number in assembly

- "counter++" has to:
  1. Copy from the memory location of the counter variable to a register
  2. Increment the register's value
  3. Copy from the register back to memory
- Assuming that "counter" is in memory location 0x8049a1c:

```
mov 0x8049a1c, %eax
add $0x1, %eax
mov %eax, 0x8049a1c
```

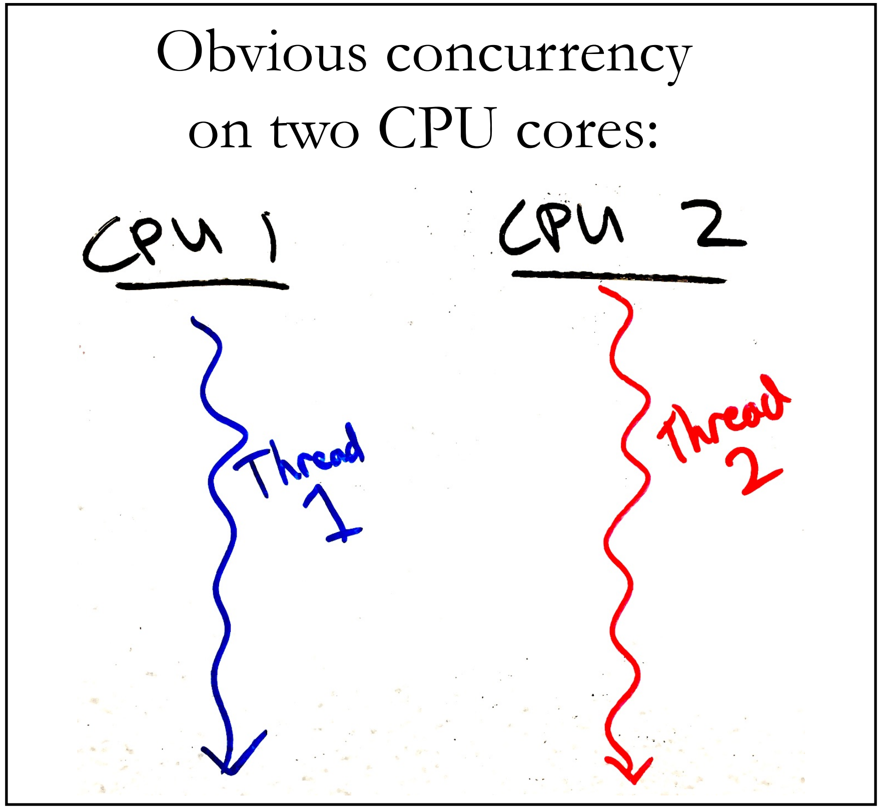
- The scheduler can interrupt the thread before or after the "add"
  - This would cause both threads to ***read the same value***, increment it to the same value, and thus they would **repeat work**.

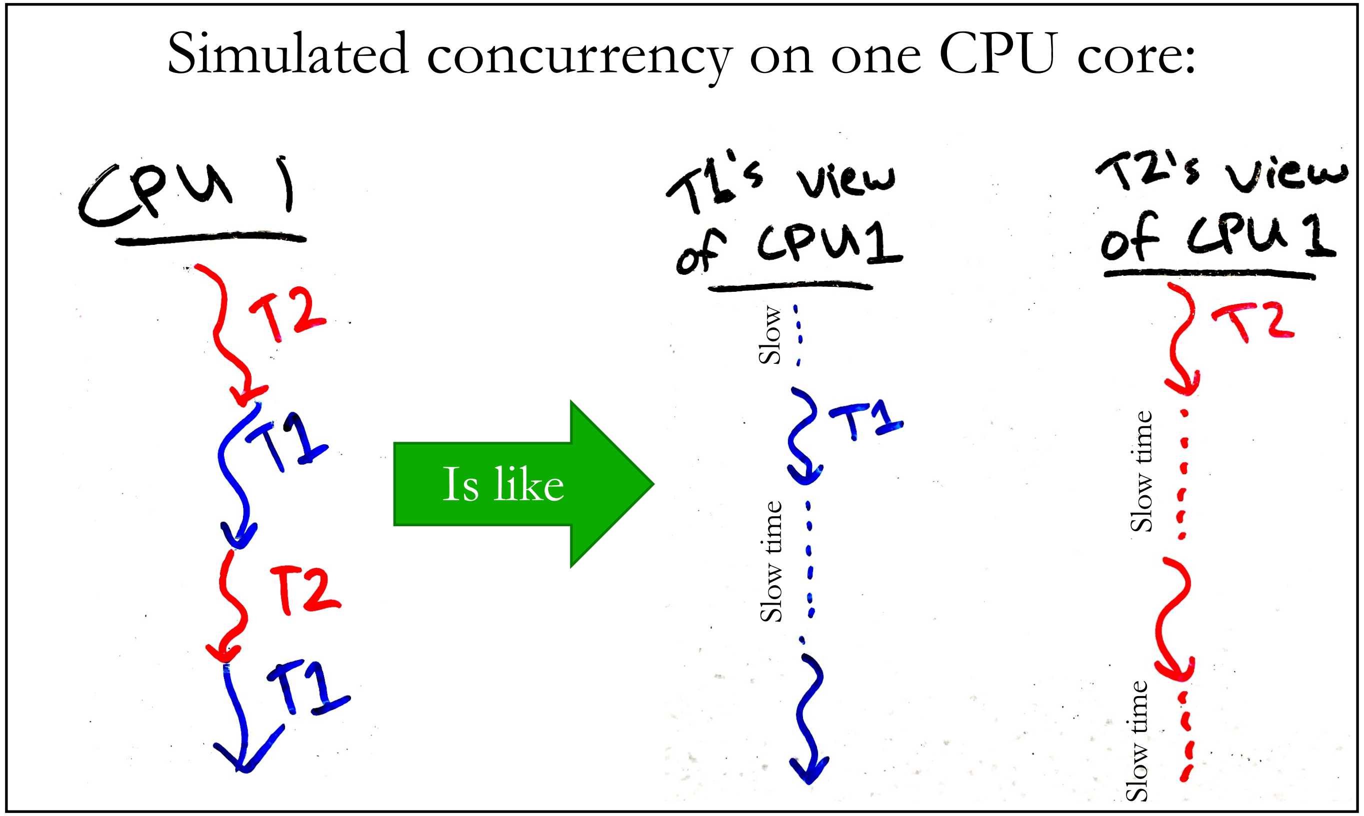
The increment failure in detail: 50 + 1 + 1 = 51!

			(after instruction)		
OS	**Thread 1**	**Thread 2**	**PC**	**%eax**	**counter**
	before critical section		100	0	50
	mov 0x8049a1c, %eax		105	**50**	50
	add $0x1, %eax		108	**51**	50
interrupt					
save T1's state					
restore T2's state			100	0	50
		mov 0x8049a1c, %eax	105	**50**	50
		add $0x1, %eax	108	**51**	50
		mov %eax, 0x8049a1c	113	51	**51**
interrupt					
save T2's state					
restore T1's state			108	51	~~50~~
	mov %eax, 0x8049a1c		113	51	**51**

The process scheduler creates concurrency

- Even if only one CPU is present, threads operate "concurrently" because they are taking turns using the CPU.
- Each process thinks it has its own CPU that is sometimes very, very slow…





Assume the scheduler is evil

- Remember that processes have no control over the scheduler.
- So, to protect against concurrency bugs, we must assume that the scheduler can interrupt us at any time and schedule any other process.
- In other words, assume that the scheduler is ***adversarial***, and will do the worst possible scheduling.
- To prevent weird and rare concurrency bugs, your code should work correctly even when faced with an evil scheduler.

Intermission

"It's not enough that we succeed. Cats must also fail."

Terminology

- ***Race condition***:
  - Two or more things are happening at the same time,
  - it's not clear which will finish first, and
  - the result will be different depending on which finishes first.
- ***Indeterminate***: Output can be different each time (not ***deterministic***).
- ***Critical section***:
  - Code that accesses a shared resource and must not be executed concurrently.
  - In other words, code that would lead to a race condition.
  - Sometimes called a ***transaction***, especially in database systems.
  - We must execute critical sections ***atomically***, meaning that it cannot be *partially* executed. Atomic means it cannot be divided, or is executed "all or none."
- ***Mutual exclusion primitives*** are used to protect critical sections.
- ***Locks*** are the simplest kind of mutual exclusion primitive.

Critical sections

- Critical sections often involve modification of multiple related data
  - While the modifications are happening there is some inconsistency
  - The inconsistency is eventually resolved before leaving the critical section
- For example:
  - Inserting an element in the middle of a linked list
    - Two pointers must change. List is broken if just one is changed.
  - Swapping two values.
- Don't have to worry about critical sections if:
  - Operation is just one assembly instruction (CPU executes these atomically), or
  - Program is single-threaded or the particular data is not shared among threads

Buggy concurrent swap

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

static volatile char* person1;
static volatile char* person2;
static const int LOOPS = 1e4;

void* mythread(void* arg) {
  printf("%s: begin\n", (char*)arg);
  int i;
  for (i=0; i<LOOPS; i++) {
    // swap
    volatile char* tmp = person1;
    person1 = person2;
    person2 = tmp;
  }
  printf("%s: done\n", (char*)arg);
  return NULL;
}
```

```
int main(int argc, char* argv[]) {
  pthread_t p1, p2;
  person1 = "Jack";
  person2 = "Jill";
  printf("main: begin (%s, %s)\n",
         person1, person2);
  pthread_create(&p1, NULL, mythread, "A");
  pthread_create(&p2, NULL, mythread, "B");

  // wait for threads to finish
  pthread_join(p1, NULL);
  pthread_join(p2, NULL);
  printf("main: end (%s, %s)\n",
         person1, person2);
}
```

https://gist.github.com/starzia/3cfd83f736b54cae1eb856561eb5e576

Critical sections in xv6: process table in kernel/proc.c

- Why do we worry about critical sections in the kernel?
  - An OS **kernel** on a multi-core machine is like a multi-threaded process, so we must protect the kernel's critical sections.
  - A hardware interrupt can happen at any time and prempt the kernel
- Process table is a shared resource
- Proc structs have many fields
- Don't want to read a proc struct that is just partially filled-in
- Don't want to accidentally assign the same "next" pid to two processes
- Etc.

```c
// Look in the process table for an UNUSED proc.
// If found, change state to EMBRYO and initialize
// state required to run in the kernel.
// Otherwise return 0.
static struct proc*
allocproc(void)
{
  struct proc *p;
  char *sp;

  acquire(&ptable.lock);
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
    if(p->state == UNUSED)
      goto found;
  release(&ptable.lock);
  return 0;

found:
  p->state = EMBRYO;
  p->pid = nextpid++;
  release(&ptable.lock);

  // Allocate kernel stack if possible.
  if((p->kstack = kalloc()) == 0){
    p->state = UNUSED;
    return 0;
  }
  sp = p->kstack + KSTACKSIZE;
```

Critical Sections

```c
// Wait for a child process to exit and return its pid.
// Return -1 if this process has no children.
int
wait(void)
{
  struct proc *p;
  int havekids, pid;

  acquire(&ptable.lock);
  for(;;){
    // Scan through table looking for zombie children.
    havekids = 0;
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if(p->parent != proc)
        continue;
      havekids = 1;
      if(p->state == ZOMBIE){
        // Found one.
        pid = p->pid;
        kfree(p->kstack);
        p->kstack = 0;
        freevm(p->pgdir);
        p->state = UNUSED;
        p->pid = 0;
        p->parent = 0;
        p->name[0] = 0;
        p->killed = 0;
        release(&ptable.lock);
        return pid;
      }
    }
```

Critical sections in Project 2

- Most importantly, when deciding whether to deallocate a shared page
- If two processes sharing the page are killed concurrently then:
  - Both processes might think the other is still using the page and ***neither*** would free it.
  - Both processes might think they are the last to use the page and ***both*** would try to free it.
- Depending on your implementation you may or may not need a lock
- The lock on the process table may already protect your critical section.
- Project 3 also has shared memory, and this time you **must** protect critical sections.

Locks

- Locks are the simplest mutual exclusion primitive
  - Represent a resource that can be reserved and freed
- Has two main functions:
- ***Acquire/lock***:
  - Used before a critical section to **reserve** the resource
  - If the lock is free (unlocked), then lock it and proceed.
  - If the lock is already taken (someone else called *acquire/lock*), then **wait until it's free** before proceeding.
- ***Release/unlock***:
  - Used at the end of a critical section to **free** the resource
  - Allows one waiting (or future) thread to acquire the lock

Two different metaphors & etymology

Lock

- A lock is something that's designed to block access.
- Our virtual lock works as follows:
  - Anyone can **lock** or **unlock** (there is no "key").
  - Trying to lock an already-locked lock will cause you to wait until it's unlocked.
- The "lock" is actually a poor/confusing metaphor.

Token

- Holding the token gives you permission to do something.
- There is only one token.
- Thus, you:
  1. Try to **acquire** the token ("lock"). You have to wait your turn if someone else is holding it.
  2. When done, **release** the token/lock.
- The token represents exclusive access to a shared resource or a critical section.

Spinlock in xv6

- `struct spinlock` stores the state of the lock (whether or not it's acquired).
- `initlock()` initializes it (just once)
- `acquire()` proceeds if the lock is not already acquired.
    - Must *atomically* check and set a value in the struct spinlock. (details next lecture)
    - If lock is already acquired, it waits until thread releases it.
- `release()` lets another thread acquire the lock later.
    - Must remember to release the lock!

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

```c
// Look in the process table for an UNUSED proc.
// If found, change state to EMBRYO and initialize
// state required to run in the kernel.
// Otherwise return 0.
static struct proc*
allocproc(void)
{
  struct proc *p;
  char *sp;

  acquire(&ptable.lock);
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
    if(p->state == UNUSED)
      goto found;
  release(&ptable.lock);
  return 0;

found:
  p->state = EMBRYO;
  p->pid = nextpid++;
  release(&ptable.lock);
```

xv6's spinlock.[ch]

```c
// Mutual exclusion lock.
struct spinlock {
  uint locked;       // Is the lock held?

  // For debugging:
  char *name;        // Name of lock.
  struct cpu *cpu;   // The cpu holding the lock.
  uint pcs[10];      // The call stack (an array of program counters)
                     // that locked the lock.
};
```

```c
void
acquire(struct spinlock *lk)
{
  pushcli(); // disable interrupts to avoid deadlock.
  if(holding(lk))
    panic("acquire");

  // The xchg is atomic.
  // It also serializes, so that reads after acquire are not
  // reordered before it.
  while(xchg(&lk->locked, 1) != 0)
    ;

  // Record info about lock acquisition for debugging.
  lk->cpu = cpu;
  getcallerpcs(&lk, lk->pcs);
}

```

```c
// Release the lock.
void
release(struct spinlock *lk)
{
  if(!holding(lk))
    panic("release");

  lk->pcs[0] = 0;
  lk->cpu = 0;

  xchg(&lk->locked, 0);

  popcli();
}
```

Recap

- Processes can have multiple ***threads*** sharing the virtual address space
- ***Critical sections*** are block of code that must be run ***atomically***
- If unprotected, critical sections lead to ***race conditions*** that make code ***indeterminant*** – we get different results depending on timing.
- ***Locks*** are the simplest ***mutual exclusion primitive***, with two main functions:
  - **Acquire/lock** – get exclusive access to a shared resource.
  - **Release/unlock** – release the shared resource.
- Concurrency occurs naturally in multi-CPU systems
- Concurrency is created by the process scheduler in single-CPU systems
- **Next time:** how locks and other synchronization primitives are built!