CS 347M (Operating Systems Minor)

Spring 2022

Lecture 12: Threads and Concurrency

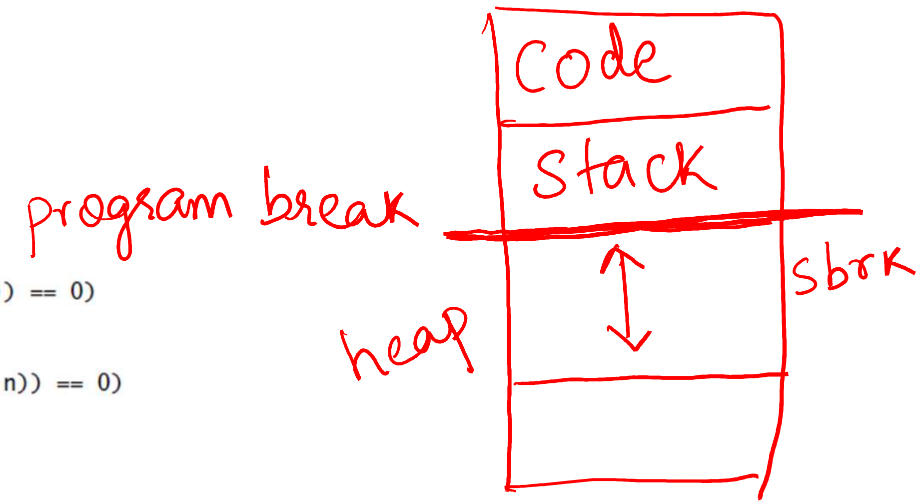
Mythili Vutukuru

CSE, IIT Bombay

Recap: Growing memory image: sbrk

- Initially heap is empty, program “break” (end of user memory) is at end of stack
  - sbrk() system call invoked by malloc to expand heap
- To grow memory, allocuvm allocates new pages, adds mappings into page table
- Whenever page table updated, must update cr3 register and TLB (done even during context switching)

```
int
growproc(int n)
{
  uint sz;
  struct proc *curproc = myproc();

  sz = curproc->sz;
  if(n > 0){
    if((sz = allocuvm(curproc->pgdir, sz, sz + n)) == 0)
      return -1;
  } else if(n < 0){
    if((sz = deallocuvm(curproc->pgdir, sz, sz + n)) == 0)
      return -1;
  }
  curproc->sz = sz;
  switchuvm(curproc);
  return 0;
}
```



allocuvm: grow address space

- Walk through new virtual addresses to be added in page size chunks
- Allocate new page, add it to page table with suitable user permissions
- Similarly deallocuvm shrinks memory image, frees up pages

```
int
allocuvm(pde_t *pgdir, uint oldsz, uint newsz)
{
  char *mem;
  uint a;

  if(newsz >= KERNBASE)
    return 0;
  if(newsz < oldsz)
    return oldsz;

  a = PGROUNDUP(oldsz);
  for(; a < newsz; a += PGSIZE){
    mem = kalloc();
    if(mem == 0){
      cprintf("allocuvm out of memory\n");
      deallocuvm(pgdir, newsz, oldsz);
      return 0;
    }
    memset(mem, 0, PGSIZE);
    if(mappages(pgdir, (char*)a, PGSIZE, V2P(mem), PTE_W|PTE_U) < 0){
      cprintf("allocuvm out of memory (2)\n");
      deallocuvm(pgdir, newsz, oldsz);
      kfree(mem);
      return 0;
    }
  }
  return newsz;
}
```

add PTE

Functions to build/walk page table

- Page table entries added by “mappages”
  - Arguments: page directory, range of virtual addresses, physical addresses to map to, permissions of the pages
  - For each page, walks page table, get pointer to PTE via function “walkpgdir”, fills it with physical addr, permissions
- Function “walkpgdir” walks page table, returns PTE of a virtual address
  - Can allocate inner page table if it doesn’t exist (based on value of last argument)

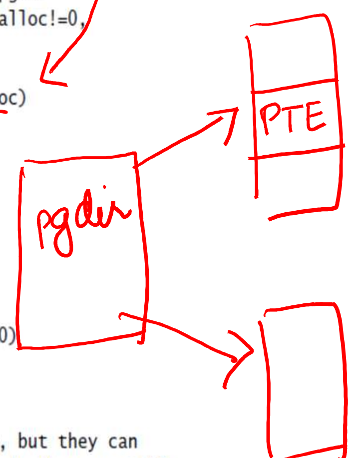
```
// Create PTEs for virtual addresses starting at va that refer to
// physical addresses starting at pa. va and size might not
// be page-aligned.
static int
mappages(pde_t *pgdir, void *va, uint size, uint pa, int perm)
{
  char *a, *last;
  pte_t *pte;

  a = (char*)PGROUNDDOWN((uint)va);
  last = (char*)PGROUNDDOWN(((uint)va) + size - 1);
  for(;;){
    if((pte = walkpgdir(pgdir, a, 1)) == 0)
      return -1;
    if(*pte & PTE_P)
      panic("remap");
    *pte = pa | perm | PTE_P;
    if(a == last)
      break;
    a += PGSIZE;
    pa += PGSIZE;
  }
  return 0;
}
```

VA to PA

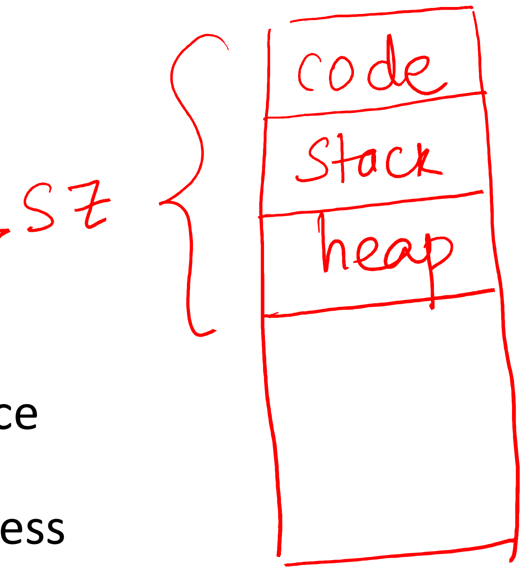
find PTE

add PA to PTE

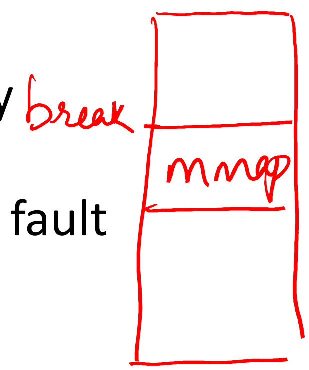
```
// Return the address of the PTE in page table pgdir
// that corresponds to virtual address va.  If alloc!=0,
// create any required page table pages.
static pte_t *
walkpgdir(pde_t *pgdir, const void *va, int alloc)
{
  pde_t *pde;
  pte_t *pgtab;

  pde = &pgdir[PDX(va)];
  if(*pde & PTE_P){
    pgtab = (pte_t*)P2V(PTE_ADDR(*pde));
  } else {
    if(!alloc || (pgtab = (pte_t*)kalloc()) == 0)
      return 0;
    // Make sure all those PTE_P bits are zero.
    memset(pgtab, 0, PGSIZE);
    // The permissions here are overly generous, but they can
    // be further restricted by the permissions in the page table
    // entries, if necessary.
    *pde = V2P(pgtab) | PTE_P | PTE_W | PTE_U;
  }
  return &pgtab[PTX(va)];
}
```



Programming assignment 2

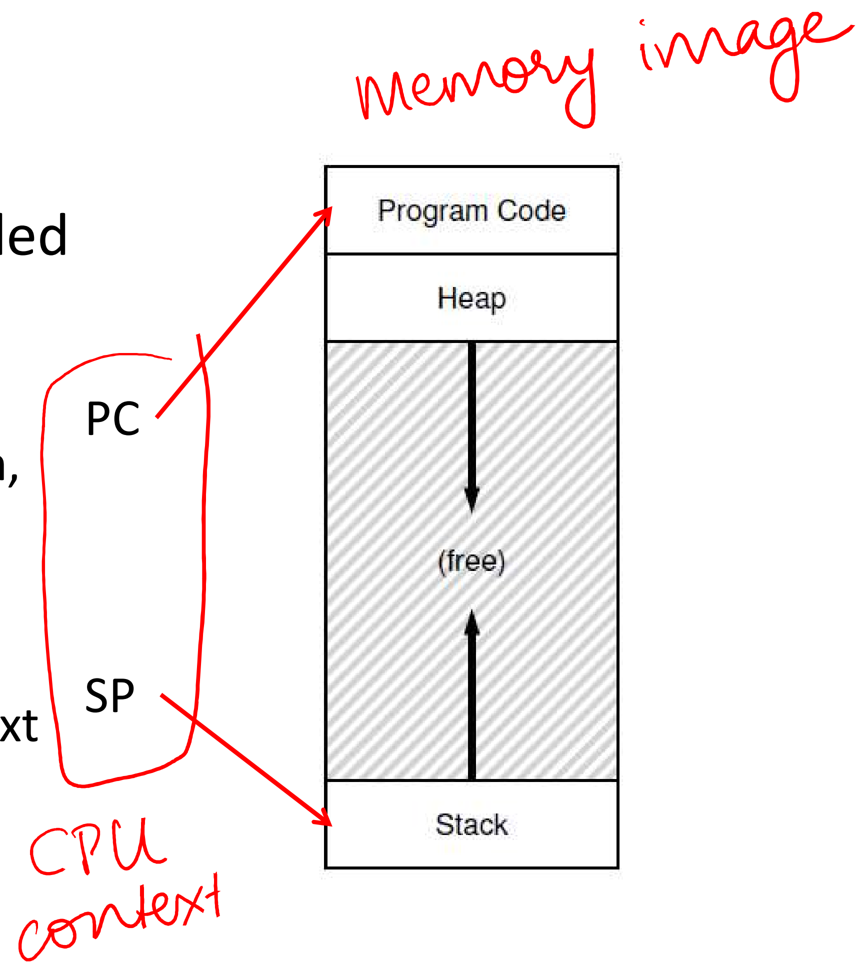


- New system calls in xv6:
  - Calculate number of pages in virtual address space
    - Process size in PCB / PGSIZE
  - Calculate number of physical pages given to process
    - Walk page table, see how many pages have present flag set
- New syscall: memory map a page and allocate memory on demand
  - Much like sbrk, add pages at program break
  - Currently sbrk system call increase both virtual and physical memory
  - Your new mmap syscall should only increase virtual memory
  - Physical memory should be allocated when page accessed and page fault
  - Page fault handler should allocate physical frame



Processes and threads

- So, far we have studied single threaded programs
- Recap: process execution
  - CPU executes instruction by instruction, traps to OS as needed
  - PC points to next instruction to run
  - SP points to current top of stack
  - Other registers also with process context
- A program can also have multiple threads of execution
- What is a thread?

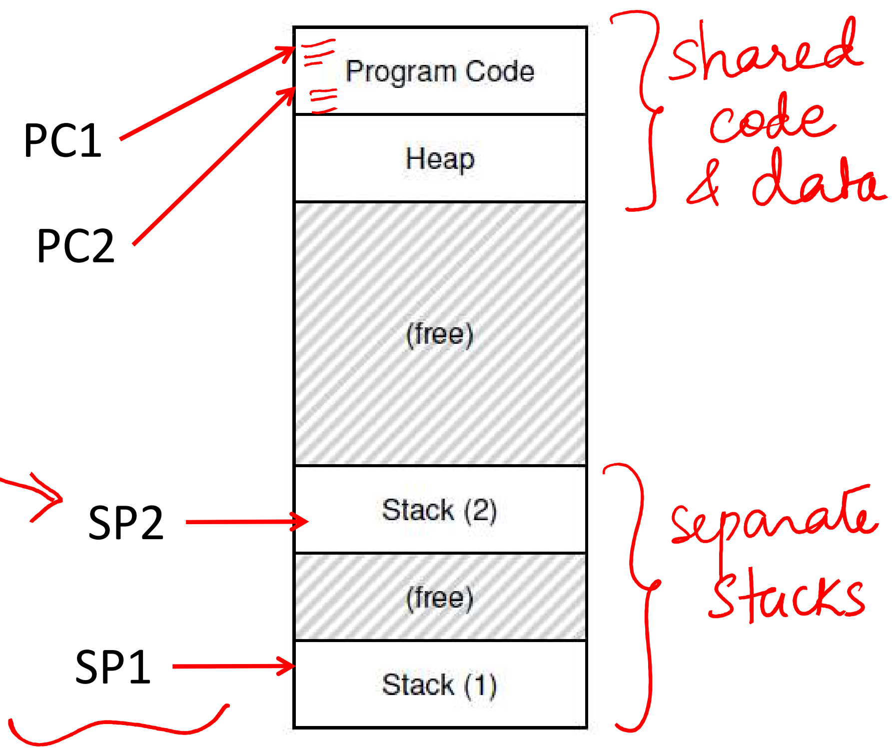


What are threads?

- Threads = light weight processes
- Why? A process may want to run multiple copies of itself
  - If one copy blocks due to blocking system call, another copy can still run
  - Multiple copies can run in parallel on multiple CPU cores
- Why not have multiple child processes running the same program?
  - Disadvantage: too much memory consumed by identical memory images
- A process can create multiple threads (default: single thread)
  - Multiple threads share same memory image of process, saves memory
  - Threads run independently on same code (if one blocks, another can still run)
  - Threads can run in parallel on multiple cores at same time

Multi-threaded process

- A thread is like another copy of a process that executes independently from parent
- Threads shares the same code, global/static data, heap
- Each thread has separate stack for independent function calls
- Each thread has separate PC
  - Each thread may run over different part of the program independently
- Each thread has separate CPU context during execution



Concurrency vs. parallelism

T1 T2 T1 T2

T1 T2

core 1 core 2

- Understand the difference between concurrency and parallelism
  - Concurrency: running multiple threads/processes at the same time, even on single CPU core, by interleaving their executions
  - Parallelism: running multiple threads/processes in parallel over different CPU cores
- With multiple threads, process can get better performance on multicore systems via parallelism
- Even if no parallelism (single core), concurrency of threads ensures effective use of CPU when one of the threads blocks (e.g., for I/O)

POSIX threads

- In Linux, POSIX threads (pthreads) library allows creation of multiple threads in a process
- Each thread is given a start function where its execution begins
  - Threads execute independently from parent after creation
  - Parent can wait for threads to finish (optional)
- Several such threading libraries exist in different programming languages

```
void f1() {
    …
}

void f2() {
    …
}

main() {
    …
    pthread_t t1, t2
    pthread_create(&t1, .., f1,..)
    pthread_create(&t2, .., f2,..)
    …

    pthread_join(t1, ..)
    pthread_join(t2, ..)

}
```

Creating threads using pthreads API

```c
#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;
    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;
}
```

Figure 26.2: **Simple Thread Creation Code (t0.c)**

Scheduling threads

- OS schedules threads that are ready to run independently, like processes
- The context of a thread (PC, registers) is saved into/restored from thread control block (TCB)
  - Every PCB has one or more linked TCBs
- Threads that are scheduled independently by kernel are called kernel threads
  - E.g., Linux pthreads are kernel threads
- In contrast, some libraries provide user-level threads
  - User program sees multiple threads, but kernel is aware of fewer threads
  - Multiple such user threads may not be scheduled in parallel by kernel
  - Why use user threads then? Ease of programming

Example: threads with shared data

- Shared global counter
- Two threads update same counter 10^7 times
- What is expected output after both threads finish?

```
static volatile int counter = 0;

//
// mythread()
//
// Simply adds 1 to counter repeatedly, in a loop
// No, this is not how you would add 10,000,000 to
// a counter, but it shows the problem nicely.
//
void *
mythread(void *arg)
{
    printf("%s: begin\n", (char *) arg);
    int i;
    for (i = 0; i < 1e7; i++) {
        counter = counter + 1;
    }
    printf("%s: done\n", (char *) arg);
    return NULL;
}

//
// main()
//
// Just launches two threads (pthread_create)
// and then waits for them (pthread_join)
//
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");

    // join waits for the threads to finish
    Pthread_join(p1, NULL);
    Pthread_join(p2, NULL);
    printf("main: done with both (counter = %d)\n", counter);
    return 0;
}
```

Threads with shared data: what happens?

- What do we expect? Two threads, each increments counter by 10^7, so 2X10^7

```
prompt> gcc -o main main.c -Wall -pthread
prompt> ./main
main: begin (counter = 0)
A: begin
B: begin
A: done
B: done
main: done with both (counter = 20000000)
```

- Sometimes, a lower value. Why?

```
prompt> ./main
main: begin (counter = 0)
A: begin
B: begin
A: done
B: done
main: done with both (counter = 19345221)
```

Understanding shared data access

load counter → reg
reg = reg + 1
store reg → counter

- The C code "counter = counter + 1" is compiled into multiple instructions
  - Load counter variable from memory into register
  - Increment register
  - Store register back into memory of counter variable
- What happens when two threads run this line of code concurrently?
  - Counter is 0 initially
  - T1 loads counter into register, increment reg
  - Context switch, register (value 1) saved
  - T2 runs, loads counter 0 from memory
  - T2 increments register, stores to memory
  - T1 resumes, stores register value to counter
  - Counter value rewritten to 1 again
  - Final counter value is 1, expected value is 2

T1

load counter → reg
reg = reg + 1
(context switch, save reg)

T2

load counter → reg
reg = reg + 1
store reg → counter

(resume, restore reg)
store reg → counter

Race conditions, critical sections

- Incorrect execution of code due to concurrency is called race condition
    - Due to unfortunate timing of context switches, atomicity of data update violated
- Race conditions happen when we have concurrent execution on shared data
    - Threads sharing common data in memory image of user processes
    - Processes in kernel mode sharing OS data structures
- We require mutual exclusion on some parts of user or OS code
    - Concurrent execution by multiple threads/processes should not be permitted
- Parts of program that need to be executed with mutual exclusion for correct operation are called critical sections
    - Present in multi-threaded programs, OS code
- How to access critical sections with mutual exclusion? Using locks (next topic)