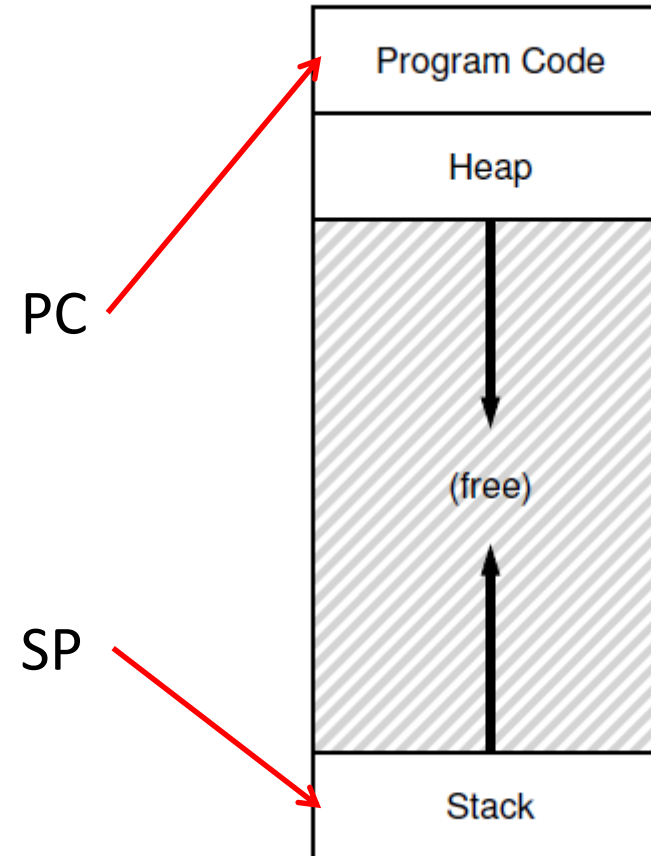


Threads and concurrency

Mythili Vutukuru
CSE, IIT Bombay

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?

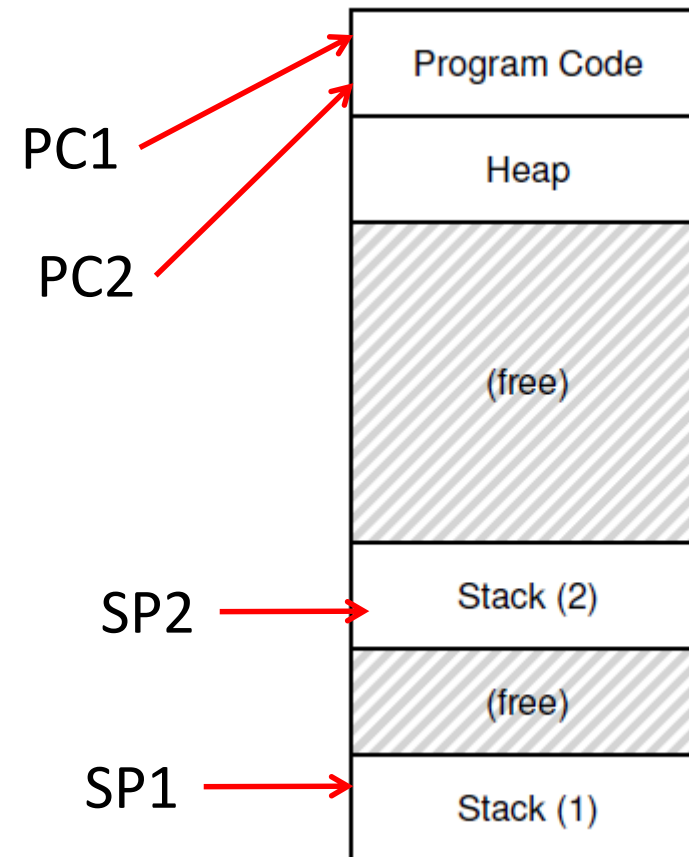


What are threads?

- 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?
 - Too much memory consumed by identical memory images
 - Needs IPC to share information across processes
- 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
 - Threads can share data more easily

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, running different code
- Each thread has separate CPU context during execution



Concurrency vs. parallelism

- 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 (**pthread**s) 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

```
1  #include <stdio.h>
2  #include <assert.h>
3  #include <pthread.h>
4
5  void *mythread(void *arg) {
6      printf("%s\n", (char *) arg);
7      return NULL;
8  }
9
10 int
11 main(int argc, char *argv[]) {
12     pthread_t p1, p2;
13     int rc;
14     printf("main: begin\n");
15     rc = pthread_create(&p1, NULL, mythread, "A"); assert(rc == 0);
16     rc = pthread_create(&p2, NULL, mythread, "B"); assert(rc == 0);
17     // join waits for the threads to finish
18     rc = pthread_join(p1, NULL); assert(rc == 0);
19     rc = pthread_join(p2, NULL); assert(rc == 0);
20     printf("main: end\n");
21     return 0;
22 }
```

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 are seen as one thread by kernel, may not be scheduled in parallel for this reason
 - 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?

```
4
5  static volatile int counter = 0;
6
7  //
8  // mythread()
9  //
10 // Simply adds 1 to counter repeatedly, in a loop
11 // No, this is not how you would add 10,000,000 to
12 // a counter, but it shows the problem nicely.
13 //
14 void *
15 mythread(void *arg)
16 {
17     printf("%s: begin\n", (char *) arg);
18     int i;
19     for (i = 0; i < 1e7; i++) {
20         counter = counter + 1;
21     }
22     printf("%s: done\n", (char *) arg);
23     return NULL;
24 }
25
26 //
27 // main()
28 //
29 // Just launches two threads (pthread_create)
30 // and then waits for them (pthread_join)
31 //
32 int
33 main(int argc, char *argv[])
34 {
35     pthread_t p1, p2;
36     printf("main: begin (counter = %d)\n", counter);
37     Pthread_create(&p1, NULL, mythread, "A");
38     Pthread_create(&p2, NULL, mythread, "B");
39
40     // join waits for the threads to finish
41     Pthread_join(p1, NULL);
42     Pthread_join(p2, NULL);
43     printf("main: done with both (counter = %d)\n", counter);
44     return 0;
45 }
```

Threads with shared data: what happens?

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

```
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)
```

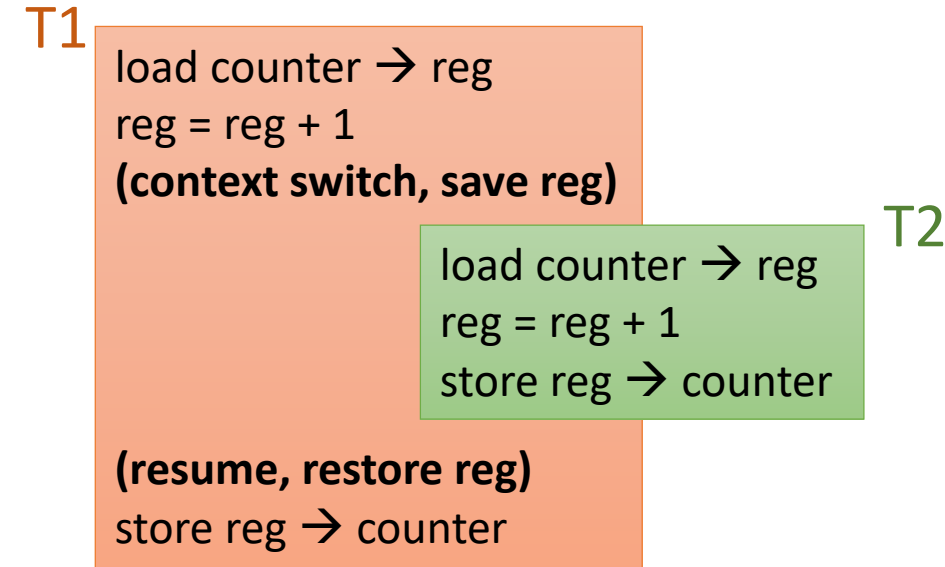
Understanding shared data access

- 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

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

Understanding shared data access

- 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



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)