Processes and Threads

CS 416: Operating Systems Design, Spring 2011

Department of Computer Science
Rutgers University

Process Control Block

OS maintains a *Process Control Block (PCB) for each process*

The PCB is a big data structure with many fields:

- Process ID
- User ID
- Execution state

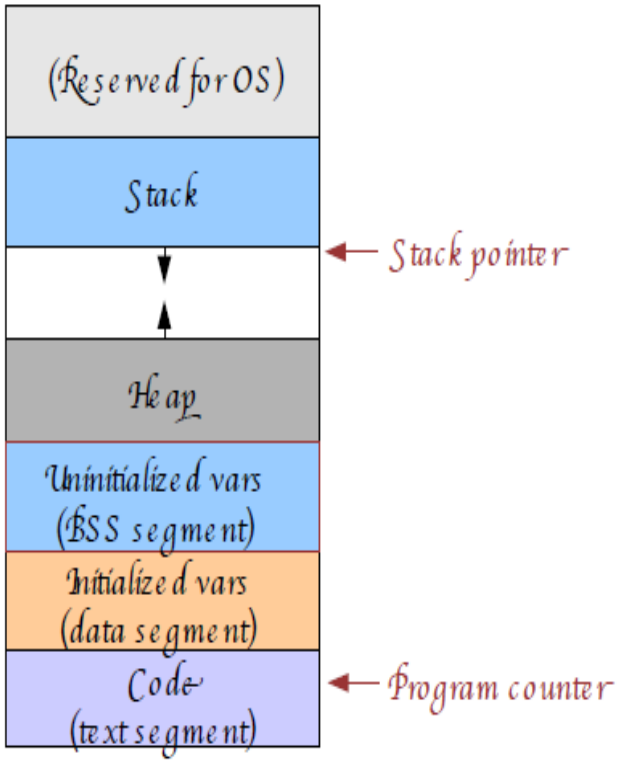
  -ready, running, or waiting
- Saved CPU state

  -CPU registers saved the last time the process was suspended.
- OS resources

  -Open files, network sockets, etc.
- Memory management info
- Scheduling priority

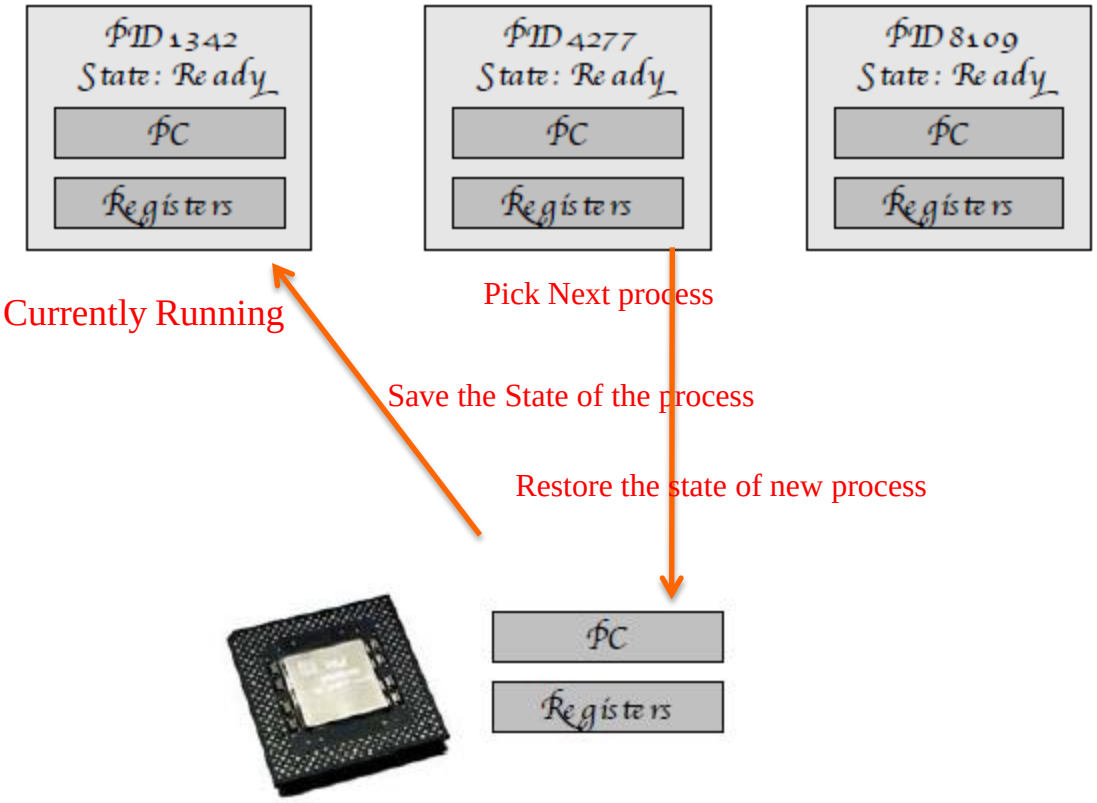
  -Give some processes higher priority than others
- Accounting information

  -Total CPU time, memory usage, etc.

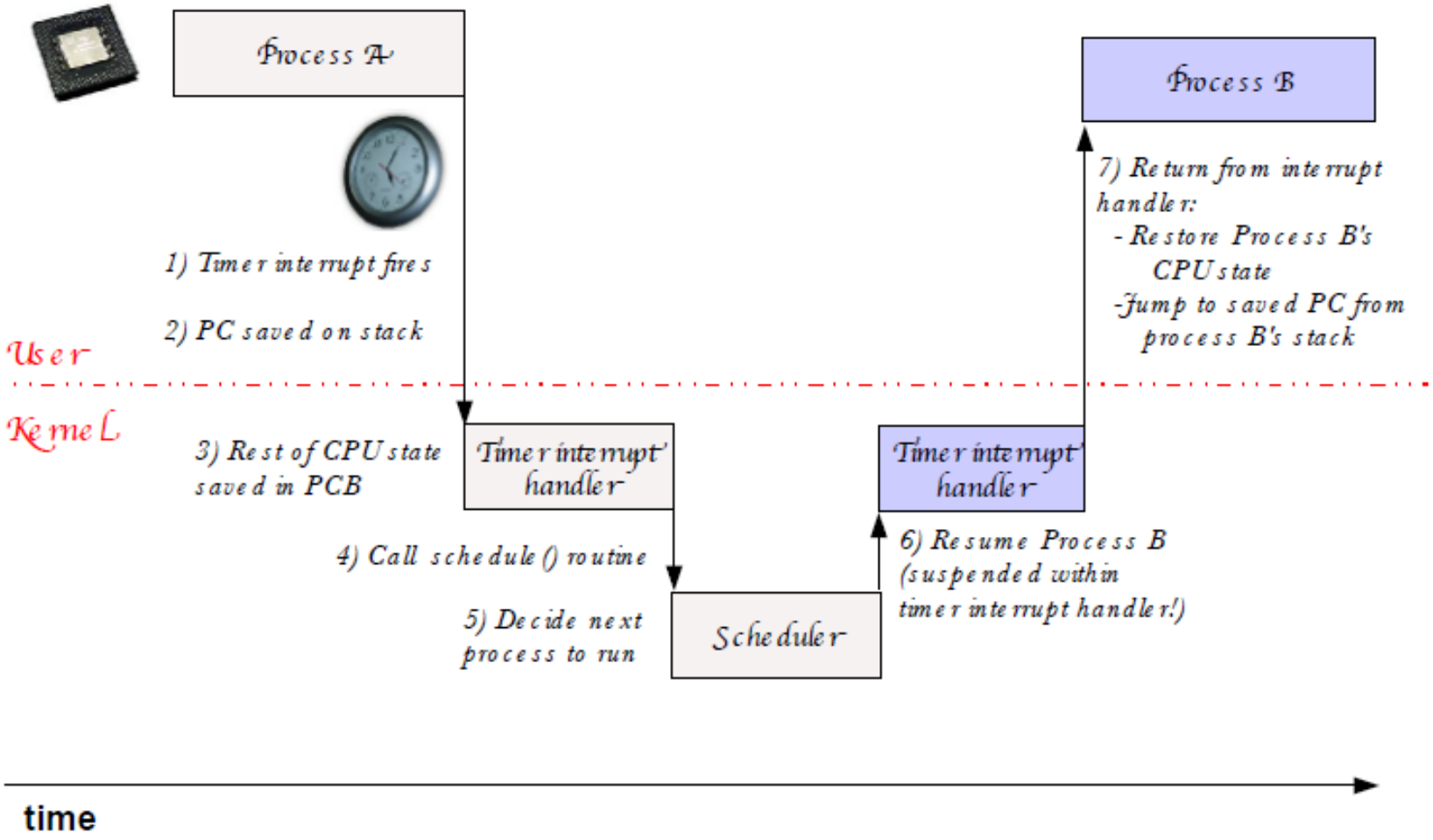


What is Context Switching

The act of swapping a process state on or off the CPU is a context switch



Context Switch in Linux



Process Creation

One process can create or fork() another process

- The Original Process is called *Parent*
- The newly created process is called *Child*

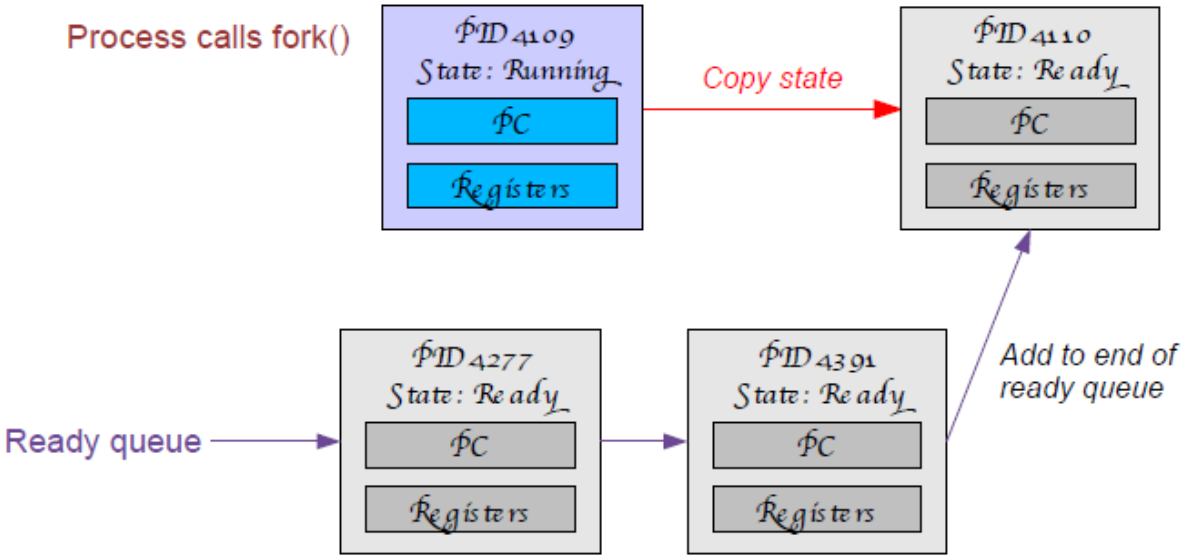
```
gayathri@gayathri-laptop:~/Courses/CS416/mytrial$ pstree -p
init(1)─┬─GoogleTalkPlugi(8582)─┬─{GoogleTalkPlug}(8583)
        │                       ├─{GoogleTalkPlug}(8584)
        │                       ├─{GoogleTalkPlug}(8585)
        │                       ├─{GoogleTalkPlug}(8586)
        │                       └─{GoogleTalkPlug}(8587)
        ├─NetworkManager(989)─┬─dhclient(3664)
        │                     └─{NetworkManager}(1890)
```

- When is the init(1) process created ?

Unix fork() mechanism

In Unix, the fork() system call is used for creating new process

- This creates an exact duplicate of the parent process
- Creates and Initializes a new PCB
- Creates a new address space
- Copies entire contents of parent's address space into the child
- Initializes the CPU and OS resources to a copy of the parents
- Places new PCB on ready queue



Unix fork mechanism

New child process starts running where fork() call returns

Child Process has an exact copy of the parent's local variable

```
#include<stdlib.h>

int main(int argc, char **argv) {
  int i,child_pid = -1;
  for (i = 0; i < 10; i++) {
    printf("Process %d: value is %d  and child_pid = %d \n", getpid(), i,child_pid);
    if (i == 5) {
        printf("Process %d: About to do a fork...\n", getpid());
        child_pid = fork();
    }
  }
return 0;
}
```

Output of the sample program

```
gayathri@gayathri-laptop:~/Courses/CS416/mytrial$ ./a.out
Process 10211: value is 0  and child_pid = -1
Process 10211: value is 1  and child_pid = -1
Process 10211: value is 2  and child_pid = -1
Process 10211: value is 3  and child_pid = -1
Process 10211: value is 4  and child_pid = -1
Process 10211: value is 5  and child_pid = -1
Process 10211: About to do a fork...
Process 10211: value is 6  and child_pid = 10212
Process 10211: value is 7  and child_pid = 10212
Process 10211: value is 8  and child_pid = 10212
Process 10211: value is 9  and child_pid = 10212
Process 10212: value is 6  and child_pid = 0
Process 10212: value is 7  and child_pid = 0
Process 10212: value is 8  and child_pid = 0
Process 10212: value is 9  and child_pid = 0
```

Waiting for child process

```c
#include<stdlib.h>

int main(int argc, char **argv) {
  int i,child_pid = -1, status;
  for (i = 0; i < 10; i++) {
    printf("Process %d: value is %d  and child_pid = %d \n", getpid(), i,child_pid);
    if (i == 5) {
        printf("Process %d: About to do a fork...\n", getpid());
        child_pid = fork();

        if(child_pid)
          wait(&status);
    }
  }
return 0;
}
```

Wait for any one child to terminate

pid_t wait(int *status);

Wait for a specific child

pid_t waitpid(pid_t pid, int *status, int options);

Output of the sample program

```
Process 10337: value is 0  and child_pid = -1
Process 10337: value is 1  and child_pid = -1
Process 10337: value is 2  and child_pid = -1
Process 10337: value is 3  and child_pid = -1
Process 10337: value is 4  and child_pid = -1
Process 10337: value is 5  and child_pid = -1
Process 10337: About to do a fork...
Process 10338: value is 6  and child_pid = 0
Process 10338: value is 7  and child_pid = 0
Process 10338: value is 8  and child_pid = 0
Process 10338: value is 9  and child_pid = 0
Process 10337: value is 6  and child_pid = 10338
Process 10337: value is 7  and child_pid = 10338
Process 10337: value is 8  and child_pid = 10338
Process 10337: value is 9  and child_pid = 10338
```

fork() and execve()

How do we start a new program instead of just a copy of the old?

- Use the UNIX execve() system call

  Int execve(const char *filename, char *const argv[], char *const envp[])

- filename: name of the executable file to run
- argv: command line arguments
- envp: environment variable settings (e.g., $PATH, $HOME)

- execve() replaces the address space and CPU state of the current process

Concurrent Programming

Many programs want to do many things "at once"

Opening several word documents:

- They all share the same code.

Scientific Programs:

- Process different parts of the data set on different CPUs
  - Share the memory of dataset being processed

In each case, would like to share memory across these activities

Can't we simply do this with multiple processes ?

Why processes are not ideal ?

Processes are not very efficient

- Each process has its own PCB and OS resources
- Typically high overhead for each process: e.g., 1.7 KB per task_struct on Linux!
- Creating a new process is often very expensive

Processes don't (directly) share memory

- Each process has its own address space
- Parallel and concurrent programs often want to directly manipulate the same memory (*e.g.When processing elements of a large array in parallel)*

Note: Many OS's provide some form of inter-process shared memory

- cf., UNIX shmget() and shmat() system calls
- *Still, this requires more programmer work and does not address the efficiency issues.*

Can we do better?

What can we share across all of these tasks?

- Same code – generally running the same or similar programs
- Same data
- Same privileges
- Same OS resources (files, sockets, etc.)

What is private to each task?

- Execution state: CPU registers, stack, and program counter

Key idea:

- Separate the concept of a process from a thread of control
- The process is the address space and OS resources
- Each thread has its own CPU execution state

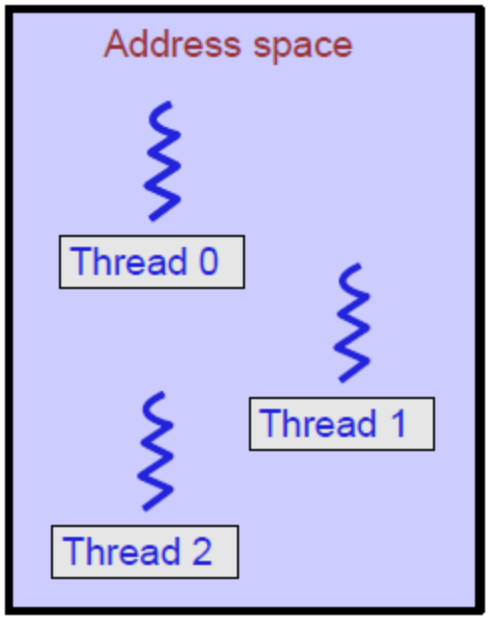
Process and Threads

Each Process has one or more threads within it

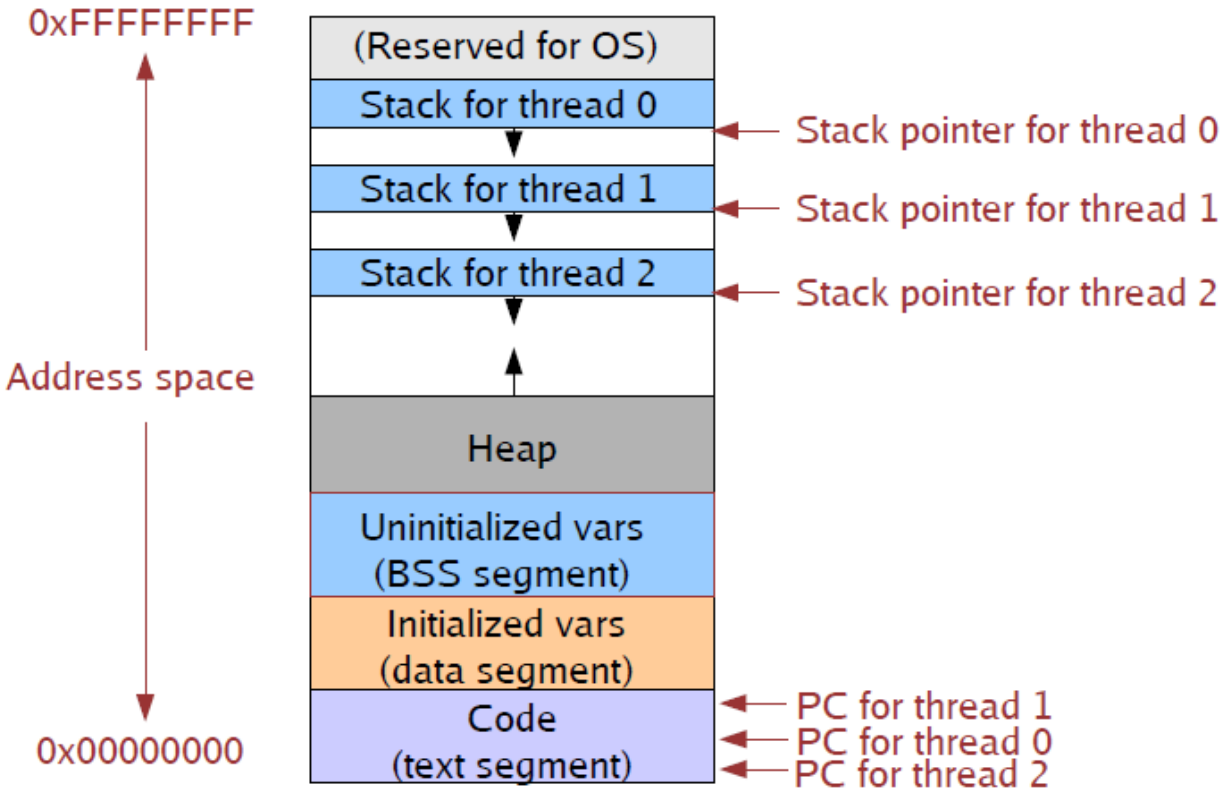
- Each thread has its own stack, CPU Registers, etc.
- All threads within a process share the same address space and OS resources
  - Threads share memory! So, they can communicate

The thread is now the unit of "CPU Scheduling"

- A process is just a container for threads
- Each thread is bound to its containing process



(New) Address Space with Threads

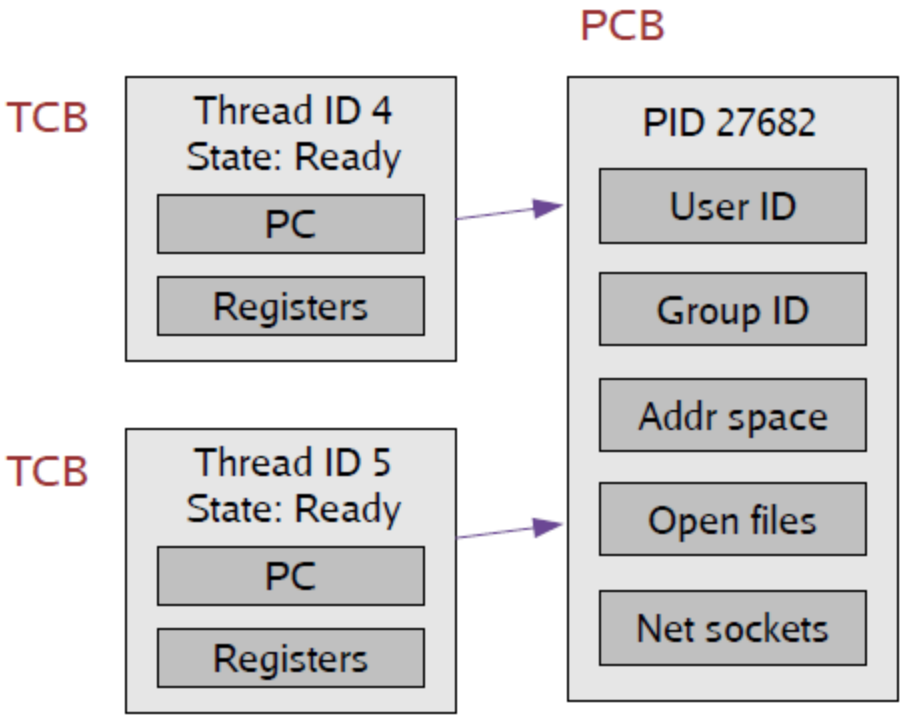


All threads in a single process share the same address space!

Implementing Threads

Idea: Break the PCB into two pieces:

- Thread Specific Stuff: CPU State
- Process Specific Stuff: Address Space and OS resources



No processor state in PCB !

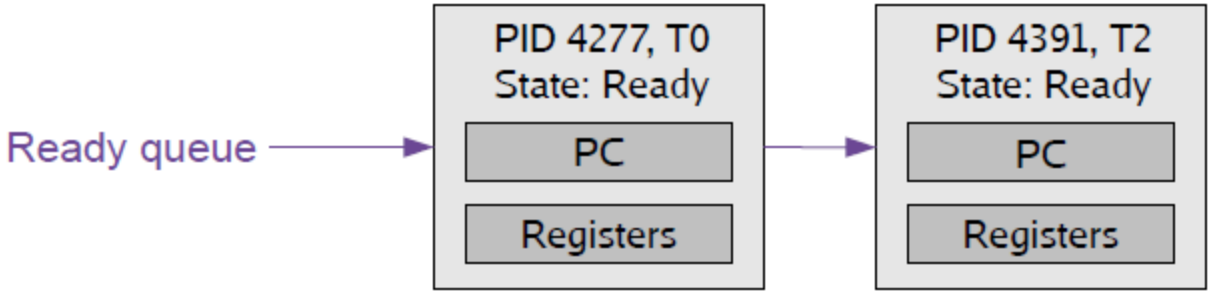
TCBs are smaller and cheaper !

TCB: 24 fields
PCB: 106 fields

Context Switching

TCB is now the unit of Context Switch

- Ready Queue, Wait Queue, etc contain pointers to TCB
- Context Switch copies CPU state to/from TCB\



Context switch between two threads of *same* process

- Need not change the address space

Context switch between two threads of *different* process

- Need to change the address space.

User Level Threads

Early UNIX systems did not support threads at the kernel threads

- OS only knew about processes with separate address space

However can still implement threads as a **User-Level library**

- OS need not know about threads

How is this possible ?

- Recall: All threads in a process share the same address space
- Managing multiple threads only requires switching CPU state (PC, Registers, etc)
- This can be done without the OS intervention (load, store instructions!)

Implementing User level Threads

Alternative to Kernel Threads

- Implement all thread functions as a user level-thread (eg. Libpthread)
- OS thinks the process has a single thread

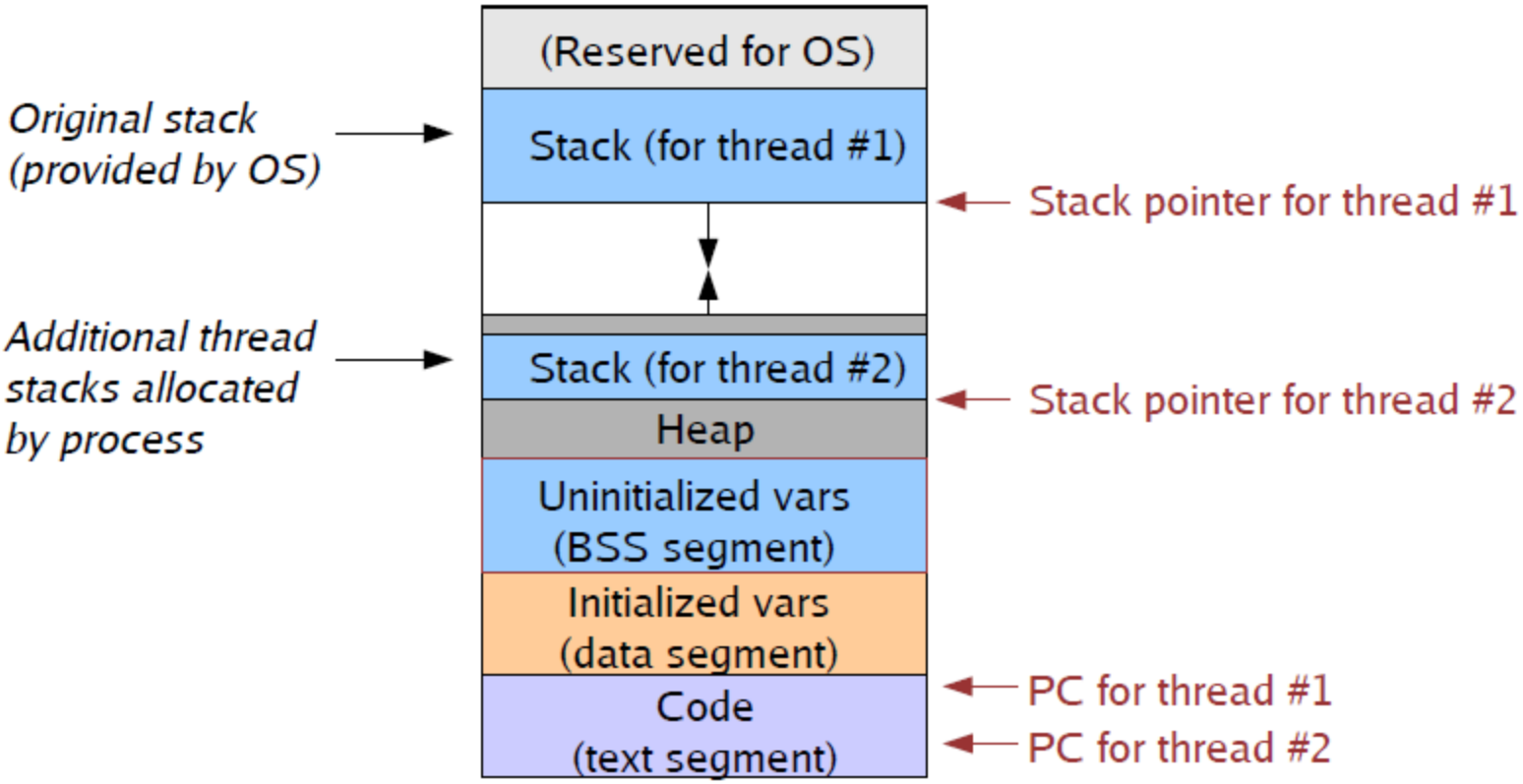
  Use the same PCB structure to represent process

How to create User-Level Thread?

- Thread library maintains a TCB for each thread in the application

  Just a linked list of TCBs
- Allocate a separate stack for each thread (usually with malloc)

User Level Thread Address Space



Stacks must be allocated carefully and managed by the thread library.

Preemptive vs non-preemptive threads

How to prevent the single user thread from hogging CPU ?

Strategy 1: Require threads to cooperate

- Each thread must call back into the thread library periodically
- Yield() -> Thread voluntarily gives up the CPU

Strategy 2: Use Preemption

- OS signals the thread library periodically (clock triggered signal)
- A signal is like a hardware interrupt
  - Causes the process to jump to signal handler
- The signal handler gives control back to the thread library
  - Thread library then context switches to new thread

Process Signals

User program invokes OS services - *System calls*

OS notifies process of an event - *Signal*

Signals

- UNIX mechanism for OS to notify a user program when an event of interest occurs
- Potentially interesting events are predefined: e.g., segmentation violation, message arrival, kill, etc.
- When interested in "handling" a particular event (signal), a process indicates its interest to the OS and gives the OS a procedure that should be invoked in the upcall
  - How does a process "indicate" its interest in handling signal ?

sighandler_t signal (int signum, sighandler_t action)

Signals (Cont'd)

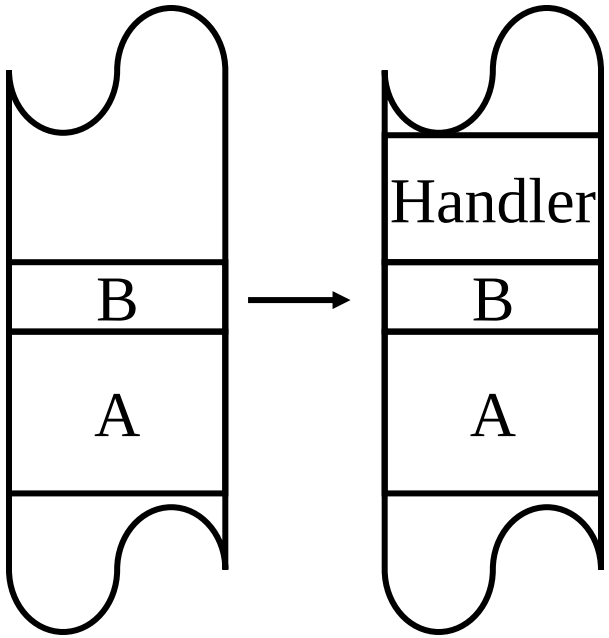
When an event of interest occurs:

The kernel handles the event first, then modifies the process's stack to look as if the process's code made a procedure call to the signal handler.

Puts an activation record on the user-level stack corresponding to the event handler

When the user process is scheduled next it executes the handler first

From the handler the user process returns to where it was when the event occurred



Threading Issues

fork()/exec()? – What if a thread issues a fork ?

Signals?

What happens if kernel wants to signal a process when all of its threads are blocked?

When there are multiple threads, which thread should the kernel deliver the signal to?

OS writes into process control block that a signal should be delivered

Next time any thread from this process is allowed to run, the signal is delivered to that thread as part of the context switch

Thread Implementation

Kernel-level threads (lightweight processes)

- Kernel sees multiple execution context
- Thread management done by the kernel

User-level threads

- Implemented as a thread library which contains the code for thread creation, termination, scheduling and switching
- Kernel sees one execution context and is unaware of thread activity
- Can be preemptive or not

User-Level vs. Kernel-Level Threads

Advantages of user-level threads

Performance: low-cost thread operations (do not require crossing protection domains)

Flexibility: scheduling can be application specific

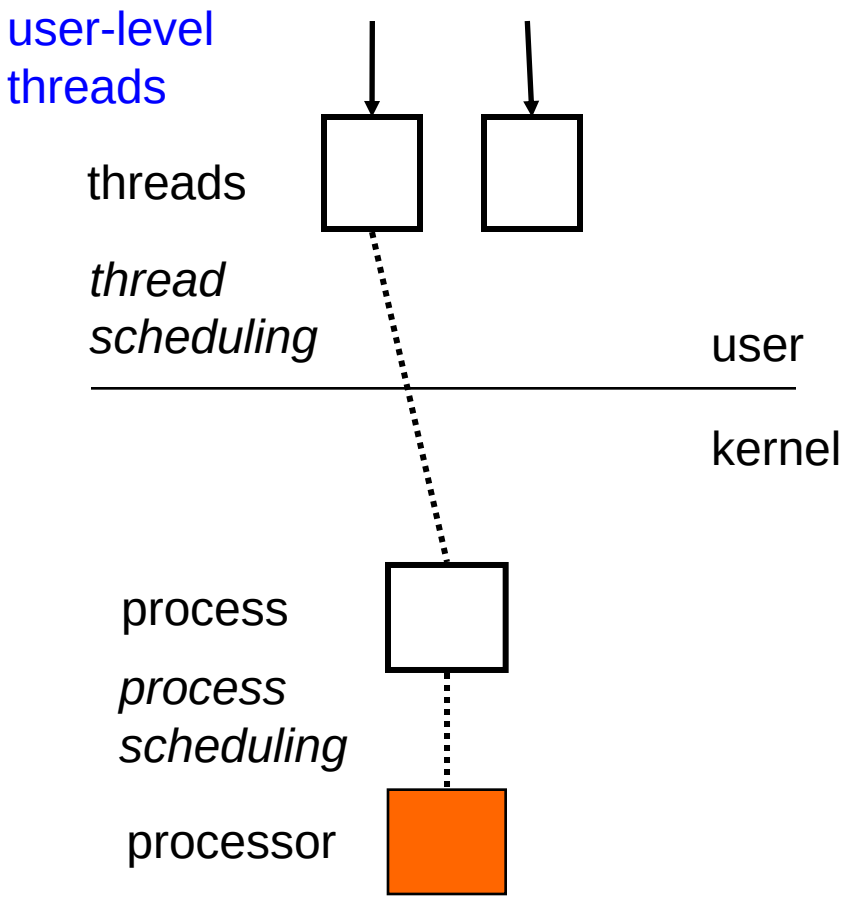
Portability: user-level thread library easy to port

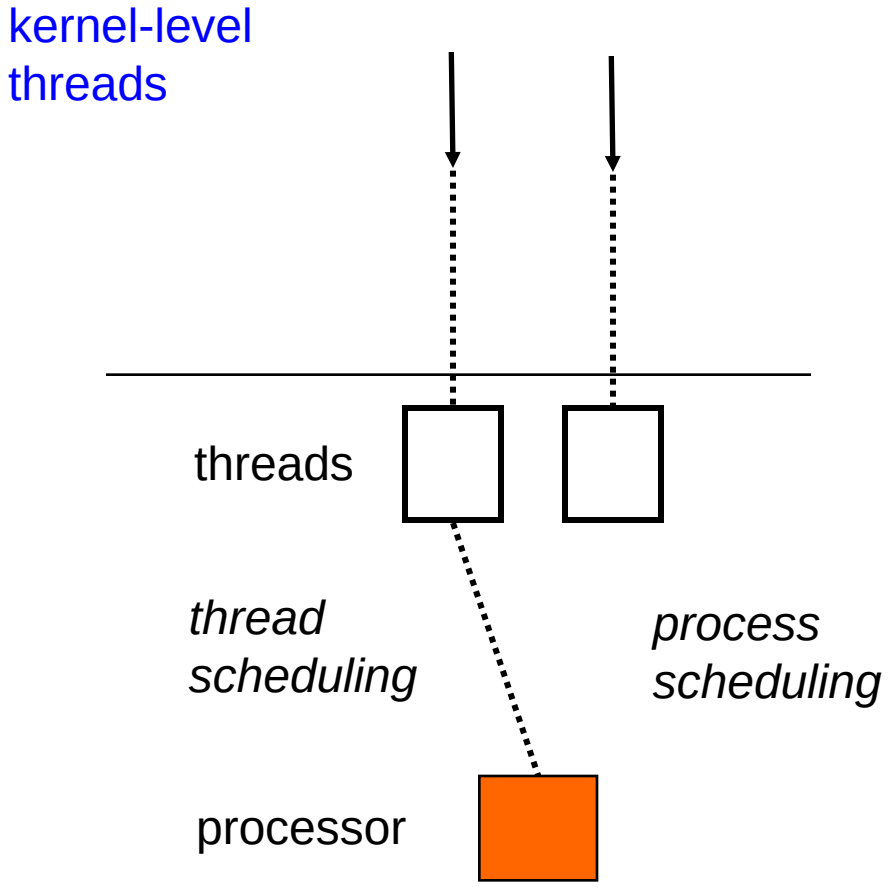
Disadvantages of user-level threads

If a user-level thread is blocked in the kernel, the entire process (all threads of that process) are blocked

Cannot take advantage of multiprocessing (the kernel assigns one process to only one processor)

User-Level vs. Kernel-Level Threads



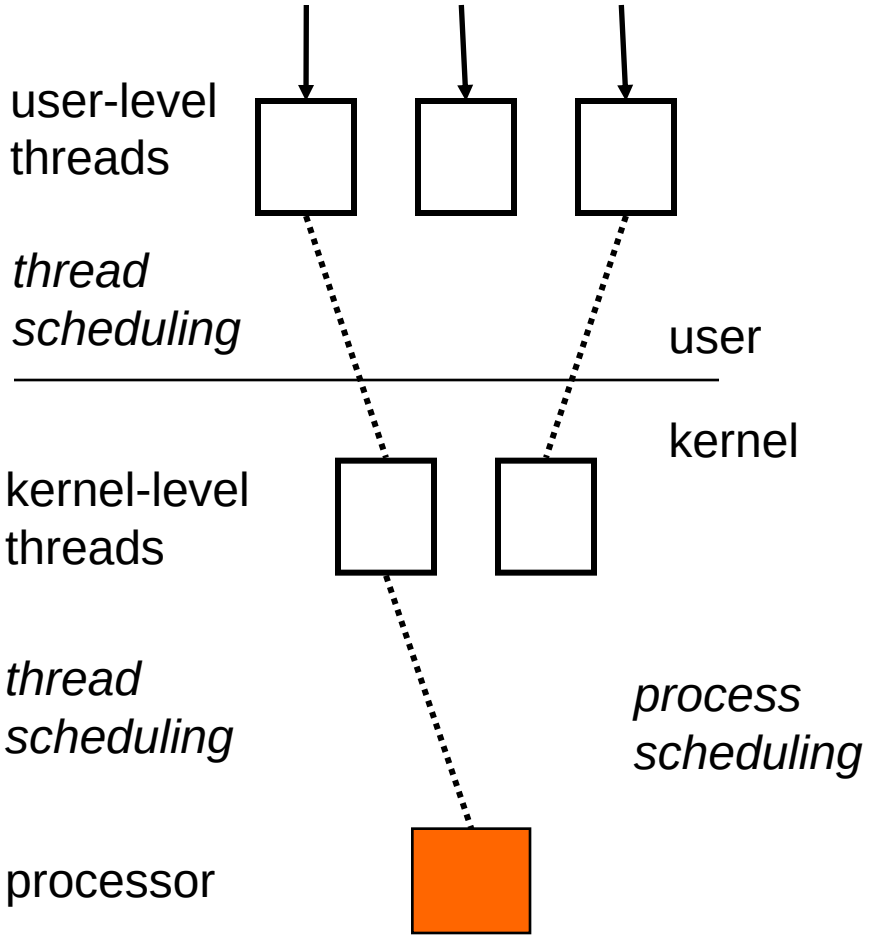


User-Level vs. Kernel-Level Threads

No reason why we shouldn't have both

Most systems now support kernel threads

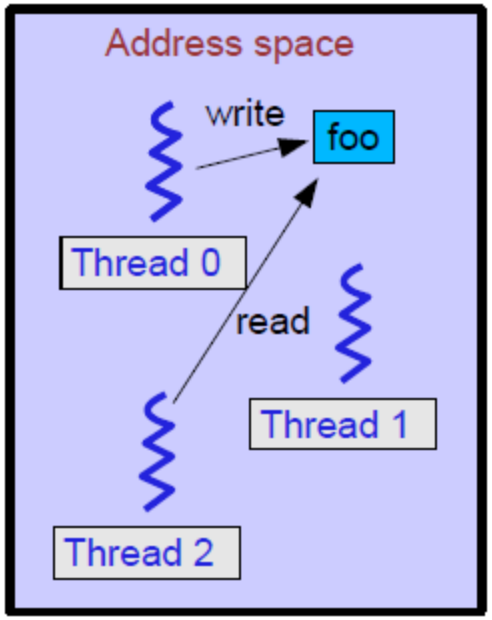
User-level threads are available as linkable libraries



More Threading Issues ?

All threads share memory

- What happens when two threads access the same variables
- Which value does thread-2 see when it reads foo ?
- What does it depend on ?



This leads to "**SYNCHRONIZATION**"