Introduction to Operating Systems

Chapter 2: Processes and threads

Manuel

Fall 2017

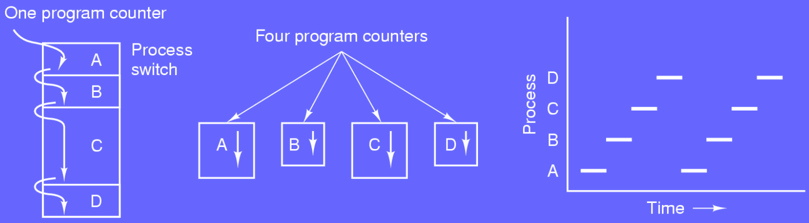
Outline

Basics

Process: abstraction of a running program

- At the core of the OS
- Process is the unit for resource management
- Oldest and most important concept
- Turn a single CPU into multiple virtual CPUs
- CPU quickly switches from process to process
- Each process run for 10-100 ms
- Processes hide the effect of interrupts

Multiprogramming



- CPU switches rapidly back and forth among all the processes
- Rate of computation of a process is not uniform/reproducible
- Potential issue under time constraints; e.g.
    - Read from tape
    - Idle loop for tape to get up to speed
    - Switch to another process
    - Switch back…too late,

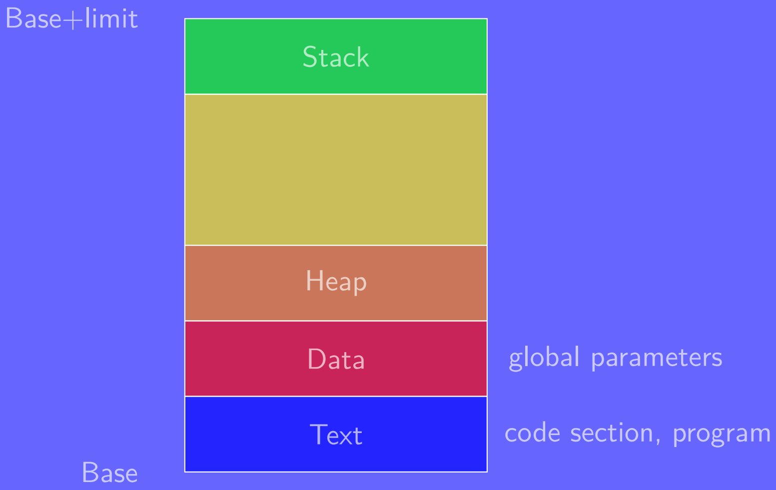
Program vs. process

Differences between programs and processes:

- Running twice a program generates two processes
- Program: sequence of operations to perform
- Process: program, input, output, state

e.g. describe process of baking a cake

Process in memory



Process creation

Four main events causing process to be created:

- System initialization
- Execution of a "process creation" system call
- A user requests a new process
- Initiation of a batch job

Example

Unix like systems:

- Creating a new process is done using one system call: `fork()`
- The call creates an exact clone of the calling process
- Child process executes `execve` to run a new program

Windows system:

- Function call `CreateProcess`, creates a new process and loads the program in the new process
- This call takes 10 parameters

Parent and child have their own address space and a change in one is invisible to the other

Process termination

Any created processes ends at some stage:

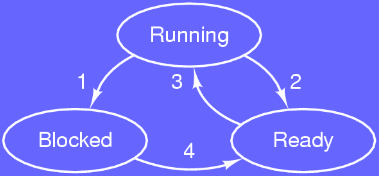
- Normal exit (voluntary)
  work is done, execute a system call to tell OS it is finished
  `exit`, `ExitProcess`
- Error exit (voluntary)
  e.g. requested file does not exist
- Fatal error (involuntary)
  e.g. referencing non existent memory, dividing by 0
- Killed by another process (involuntary)
  `kill`, `TerminateProcess`

Process hierarchies

Two main approaches:

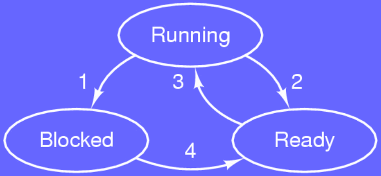
- UNIX like systems:
  - Parent creates a child
  - Child can create its own child
  - The hierarchy is called *process group*
  - Impossible to disinherit a child
- Windows system:
  - All processes are equal
  - A parent has a token to control its child
  - Token can be given to another process

Process states



1. Waiting for some input
2. Scheduler picks another process
3. Scheduler picks this process
4. Input becomes available

Process states



1. Waiting for some input
2. Scheduler picks another process
3. Scheduler picks this process
4. Input becomes available

Change of perspective on the inside of the OS:

- Do not think in terms of interrupt but of process
- Lowest level of the OS is the scheduler
- Interrupt handling, starting/stopping processes are hidden in the scheduler

Modelling processes

- Each process is represented using a structure called *process control block*
- Structure contains important information such as:
  - State
  - Program counter
  - Stack pointer
  - Memory allocation
  - Open files
  - Scheduling information
  - ...
- All the processes are stored in an array called *process table*

Modelling processes

Structure

Process management	Memory management	File management
registers	pointer to text segment info	root directory
program counter	pointer to data segment info	working directory
program status word	pointer to stack segment info	file descriptors
stack pointer		user ID
process state		group ID
priority		
scheduling parameters		
process ID		
parent process		
process group		
signals		
starting time		
CPU time used		
children's CPU time		
next alarm		

Interrupts and processes

Lowest OS level:

1. Push user program counter, PSW. . . on stack
2. Hardware loads new program counter from interrupt vector
3. Save registers (assembly)
4. Setup new stack (assembly)
5. Finish up the work for the interrupt
6. Scheduler decides which process to run next
7. Load and run the “new current process”, i.e. memory map, registers. . . (assembly)

Outline

Overview

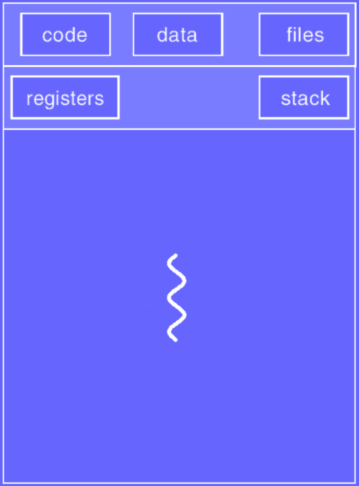
Thread: basic unit of CPU utilisation consisting of:

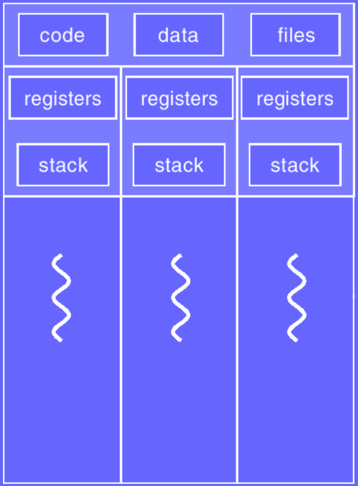
- Thread ID
- Program counter
- Register set
- Stack space

All the threads within a process share:

- Code section
- Data section
- OS resources

Single vs. multi-threaded

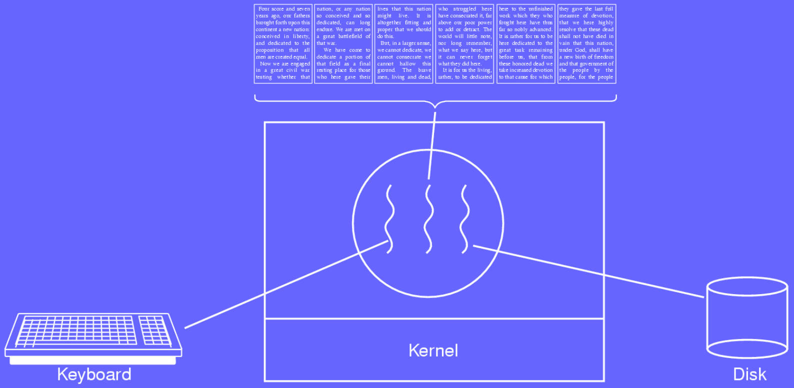




Notes and remarks

- Thread can be in the same states as a process
- Transitions are similar to the case of a process
- Threads are sometimes called lightweight process
- No protection required for threads, compared to processes
- A process starts with one threads and can create some more
- Processes want as much CPU as they can
- Threads can give up the CPU to let others using it

Example



Four score and seven years ago, our fathers brought forth upon this continent a new nation: conceived in liberty, and dedicated to the proposition that all men are created equal.
Now we are engaged in a great civil war testing whether that
nation, or any nation so conceived and so dedicated, can long endure. We are met on a great battlefield of that war.
We have come to dedicate a portion of that field as a final resting place for those who here gave their
lives that this nation might live. It is altogether fitting and proper that we should do this.
But, in a larger sense, we cannot dedicate, we cannot consecrate we cannot hallow this ground. The brave men, living and dead,
who struggled here have consecrated it, far above our poor power to add or detract. The world will little note, nor long remember, what we say here, but it can never forget what they did here.
It is for us the living, rather, to be dedicated
here to the unfinished work which they who fought here have thus far so nobly advanced. It is rather for us to be here dedicated to the great task remaining before us, that from these honored dead we take increased devotion to that cause for which
they gave the last full measure of devotion, that we here highly resolve that these dead shall not have died in vain that this nation, under God, shall have a new birth of freedom and that government of the people by the people, for the people
Kernel
Keyboard
Disk

Multi-threading problems

Thread share many data structure:

- A thread could close a file that another thread is reading
- A thread notices a lack of memory and allocate more. A thread switch occurs, the new threads also notices the lack of memory and also allocates some

Should a child inherit all the threads form its parents?

- No: the child might not function properly
- Yes: if a parent thread was waiting for some keyboard input, who gets it?

Outline

POSIX threads

pthread.h: over 60 function calls; most important ones:

- Create a thread:
  `int pthread_create(pthread_t *thread, const pthread_attr_t *attr,void *(*start_routine) (void *), void *arg);`
- Terminate a thread:
  `void pthread_exit(void *retval);`
- Wait for a specific thread to end:
  `int pthread_join(pthread_t thread, void **retval);`
- Release CPU to let another thread run:
  `int pthread_yield(void);`
- Create and initialise a thread attribute structure:
  `int pthread_attr_init(pthread_attr_t *attr);`
- Delete a thread attribute object:
  `int pthread_attr_destroy(pthread_attr_t *attr);`

Exercise

Write a short program which creates 10 threads, each thread printing its ID and exiting.

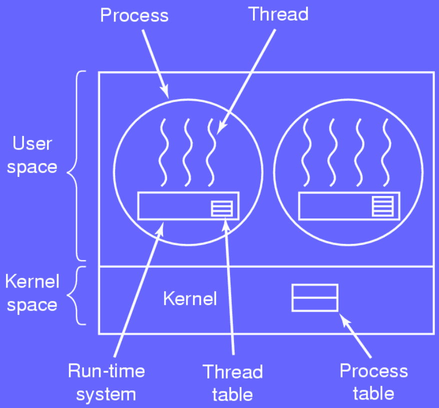
Solution

threads.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#define THREADS 10
void *gm(void *tid) {
  printf("Good morning from thread %lu\n",*(unsigned long int*)tid);
  pthread_exit(NULL);
}
int main () {
  int status, i;
  pthread_t threads[THREADS];
  for(i=0;i< THREADS;i++) {
    printf("thread %d\n",i);
    status=pthread_create(&threads[i],NULL,gm,(void*)&(threads[i]));
    if(status!=0) {
      fprintf(stderr,"thread %d failed with error %d\n",i,status);
      exit(-1);
    }
  }
  exit(0);
}
```

Threads in user space

N:1

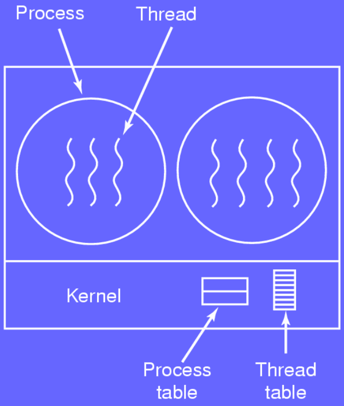


- Kernel thinks it manages single threaded processes
- Threads implemented in a library
- Thread table similar to process table, managed by run-time system
- Switching thread does not require to trap the kernel

- Problem 1: what if a thread issues a blocking system call?
- Problem 2: threat within the process have to voluntarily give up the CPU (no clock interrupt within a process)

Thread in the kernel

1:1

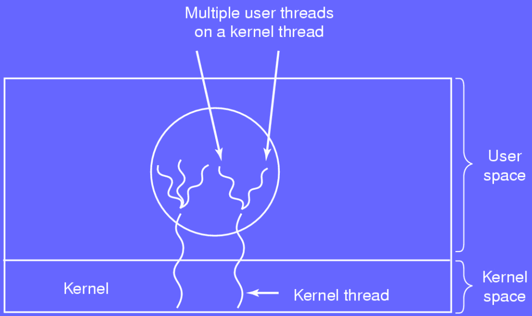


- Kernel manages the thread table
- Kernel calls are issued to request a new thread
- Calls that might block a thread are implemented as system call
- Kernel can run another thread in the meantime

- Problem 1: much higher cost than user space version
- Problem 2: signals are sent to processes not threads, which thread should handle a received signal?

Hybrid threads

M:N



- Compromise between user-level and kernel-level
- Threading library schedules user threads on available kernel threads

- Problem 1: very complex to implement
- Problem 2: scheduling often not optimal, or very expensive

Notes and remarks

- Hybrid seems attractive
- Most systems are coming back to 1:1
- Different approaches exist on how to use threads
  e.g. thread bocks on "receive system call" vs. pop up threads
- Switching implementation from single thread to multiple thread is not easy task
- Requires redesigning the whole system
- Backward compatibility must be preserved
- Research still going on to find better ways to handle threads

Key points

- What is a process?
- How can processes be created and terminated?
- What are the possible states of a process?
- What is the difference between single thread and multi-threads?
- What approaches can be taken to handle threads?

Thank you!