Operating Systems

Wang Xiaolin
wx672ster+os@gmail.com

September 13, 2019

Textbooks

- TANENBAUM A S. *Modern Operating Systems*. 3rd ed. Prentice Hall Press, 2007.
- Silberschatz, Galvin, Gagne. *Operating System Concepts Essentials*. 1st ed. John Wiley & Sons, 2011.
- BOVET D, CESATI M. *Understanding The Linux Kernel*. 3rd ed. OReilly, 2005.
- VENKATESH B, ANGRAVE L, et Al. *CS241 System Programming Coursebook*. University of Illinois, 2019.
- 郑钢. 操作系统真象还原.人民邮电出版社, 2016.
- 新设计团队. *Linux*内核设计的艺术.机械工业出版社, 2013.
- 于渊. *Orange'S*：一个操作系统的实现.电子工业出版社, 2009.

Course Web Links

Course web site: https://cs6.swfu.edu.cn/moodle

Lecture slides: https://cs2.swfu.edu.cn/~wx672/lecture_notes/os/slides/

Source code: https://cs2.swfu.edu.cn/~wx672/lecture_notes/os/src/

Lab instructions https://cs2.swfu.edu.cn/~wx672/lecture_notes/os/lab.html

Sample lab report: https://cs2.swfu.edu.cn/~wx672/lecture_notes/os/sample-report/

/etc/hosts

```
202.203.132.241  cs6.swfu.edu.cn
202.203.132.242  cs2.swfu.edu.cn
202.203.132.245  cs3.swfu.edu.cn
```

System Programming https://github.com/angrave/SystemProgramming/wiki

Beej's Guides http://beej.us/guide/

Homework

Weekly tech question

1. What was I trying to do?
2. How did I do it? (steps)
3. The expected output? The real output?
4. How did I try to solve it? (steps, books, web links)
5. How many hours did I struggle on it?

✉ wx672ster+linux@gmail.com

𝔼 Preferably in English

in stackoverflow style

OR simply show me the tech questions you asked on any website

Part I

Introduction

1 What's an Operating System

What's an Operating System?

- *"Everything a vendor ships when you order an operating system"*
- It's the program that runs all the time
- It's a resource manager
  - Each program gets time with the resource
  - Each program gets space on the resource
- It's a control program
  - Controls execution of programs to prevent errors and improper use of the computer
- No universally accepted definition

What's in the OS?

Abstractions

To hide the complexity of the actual implementations

System Goals

Convenient vs. Efficient

- Convenient for the user — for PCs
- Efficient — for mainframes, multiusers
- UNIX
  - Started with keyboard + printer, none paid to convenience
  - Now, still concentrating on efficiency, with GUI support

History of Operating Systems

Fig. 1-2. An early batch system. (a) Programmers bring cards to 1401. (b) 1401 reads batch of jobs onto tape. (c) Operator carries input tape to 7094. (d) 7094 does computing. (e) Operator carries output tape to 1401. (f) 1401 prints output.

1945 - 1955 First generation

- vacuum tubes, plug boards

1955 - 1965 Second generation

- transistors, batch systems

1965 - 1980 Third generation

- ICs and multiprogramming

1980 - present Fourth generation

- personal computers

Multi-programming is the first instance where the OS must make decisions for the users

Job scheduling — decides which job should be loaded into the memory.

Memory management — because several programs in memory at the same time

CPU scheduling — choose one job among all the jobs are ready to run

Process management — make sure processes don't offend each other

The Operating System Zoo

- Mainframe OS
- Server OS
- Multiprocessor OS
- Personal computer OS
- Real-time OS
- Embedded OS
- Smart card OS

2 OS Services

OS Services

Like a government

Helping the users:

- User interface
- Program execution
- I/O operation
- File system manipulation
- Communication
- Error detection

Keeping the system efficient:

- Resource allocation
- Accounting
- Protection and security

A Computer System

3 Hardware

CPU Working Cycle

1. Fetch the first instruction from memory
2. Decode it to determine its type and operands
3. execute it

Special CPU Registers

Program counter (PC): keeps the memory address of the next instruction to be fetched

Stack pointer (SP): ☛ the top of the current stack in memory

Program status (PS): holds

- condition code bits
- processor state

System Bus

Address Bus: specifies the memory locations (addresses) for the data transfers

Data Bus: holds the data transfered. Bidirectional

Control Bus: contains various lines used to route timing and control signals throughout the system

Controllers and Peripherals

- Peripherals are real devices controlled by controller chips
- Controllers are processors like the CPU itself, have control registers
- Device driver writes to the registers, thus control it
- Controllers are connected to the CPU and to each other by a variety of buses

Cache bus
Local bus
Memory bus
Level 2 cache
CPU
PCI bridge
Main memory
PCI bus
SCSI
USB
ISA bridge
IDE disk
Graphics adaptor
Available PCI slot
Mon-itor
Mouse
Key-board
ISA bus
Modem
Sound card
Printer
Available ISA slot

Motherboard Chipsets

- The CPU doesn't know what it's connected to
  - CPU test bench? network router? toaster? brain implant?
- The CPU talks to the outside world through its pins
  - some pins to transmit the physical memory address
  - other pins to transmit the values
- The CPU's gateway to the world is the front-side bus

Intel Core 2 QX6600

- 33 pins to transmit the physical memory address
  - so there are 2^{33} choices of memory locations
- 64 pins to send or receive data
  - so data path is 64-bit wide, or 8-byte chunks

This allows the CPU to physically address 64GB of memory ($2^{33} \times 8B$)

Some physical memory addresses are mapped away!

- only the addresses, not the spaces
- Memory holes
  - $640KB \sim 1MB$
  - `/proc/iomem`

Memory-mapped I/O

- BIOS ROM
- video cards
- PCI cards
- …

This is why 32-bit OSes have problems using 4 gigs of RAM.

the northbridge

1. receives a physical memory request
2. decides where to route it
   - to RAM? to video card? to …?
   - decision made via the memory address map

4 Bootstrapping

Bootstrapping

Can you pull yourself up by your own bootstraps?

A computer cannot run without first loading software but must be running before any software can be loaded.

Intel x86 Bootstrapping

1. BIOS (`0xffffff0`)
   ➠ POST ➠ HW init ➠ Find a boot device (FD,CD,HD…) ➠ Copy sector zero (MBR) to RAM (`0x00007c00`)
2. MBR – the first 512 bytes, contains
   - Small code (< 446 *Bytes*), e.g. GRUB stage 1, for loading GRUB stage 2
   - the primary partition table ($16 \times 4 = 64$ *Bytes*)
   - its job is to load the second-stage boot loader.
3. GRUB stage 2 — load the OS kernel into RAM
4. 🐧 startup
5. init — the first user-space program

```
<------------- Master Boot Record (512 Bytes) ------------->
|-----//-------------------------------//-----------------|
| code area | disk-sig | null | partition table | MBR-sig |
|   440     |    4     |  2   |     16x4=64     | 0xAA55  |
|-----//-------------------------------//-----------------|
0          439        443    445              509       511
```

```
$ sudo hd -n512 /dev/sda
```

5 Interrupt

Why Interrupt?

While a process is reading a disk file, can we do...

```
while(!done_reading_a_file())
{
  let_CPU_wait();
  /* or... */
  lend_CPU_to_others();
}
operate_on_the_file();
```

Modern OS are Interrupt Driven

HW INT by sending a signal to CPU

SW INT by executing a system call

Trap (exception) is a software-generated INT coursed by an error or by a specific request from an user program

Interrupt vector is an array of pointers ☛ the memory addresses of interrupt handlers. This array is indexed by a unique device number

```
$ less /proc/devices
$ less /proc/interrupts
```

Programmable Interrupt Controllers

Interrupt Processing

Interrupt Timeline

6 System Calls

System Calls

A System Call

- is how a program requests a service from an OS kernel
- provides the interface between a process and the OS

```
$ man 2 intro
$ man 2 syscalls
```

Process management

Call	Description
pid = fork( )	Create a child process identical to the parent
pid = waitpid(pid, &statloc, options)	Wait for a child to terminate
s = execve(name, argv, environp)	Replace a process' core image
exit(status)	Terminate process execution and return status

File management

Call	Description
fd = open(file, how, ...)	Open a file for reading, writing or both
s = close(fd)	Close an open file
n = read(fd, buffer, nbytes)	Read data from a file into a buffer
n = write(fd, buffer, nbytes)	Write data from a buffer into a file
position = lseek(fd, offset, whence)	Move the file pointer
s = stat(name, &buf)	Get a file's status information

Directory and file system management

Call	Description
s = mkdir(name, mode)	Create a new directory
s = rmdir(name)	Remove an empty directory
s = link(name1, name2)	Create a new entry, name2, pointing to name1
s = unlink(name)	Remove a directory entry
s = mount(special, name, flag)	Mount a file system
s = umount(special)	Unmount a file system

Miscellaneous

Call	Description
s = chdir(dirname)	Change the working directory
s = chmod(name, mode)	Change a file's protection bits
s = kill(pid, signal)	Send a signal to a process
seconds = time(&seconds)	Get the elapsed time since Jan. 1, 1970

Program 1
Program 2
Program 3
fork()
vfork()
clone()
C Library
User space
Kernel space
System call
...
2
fork()
...
120
clone()
...
190
vfork()
...
sys_fork()
sys_clone()
sys_vfork()
do_fork()
System Call Table

Figure 1-16. How a system call can be made: (1) User program traps to the kernel. (2) Operating system determines service number required. (3) Operating system calls service procedure. (4) Control is returned to user program.

The 11 steps in making the system call `read(fd,buffer,nbytes)`

Example

Linux INT80h

Interrupt Vector Table: The very first 1KiB of x86 memory.

- 256 entries $\times$ 4B = 1KiB
- Each entry is a complete memory address (segment:offset)
- It's populated by Linux and BIOS
- Slot 80h: address of the kernel services dispatcher (☛ sys-call table)

Example

```
SECTION .DATA
Msg:      db 'Hello, world!',10  ; 10 = ascii for LF
MsgLen:   equ $-Msg

SECTION .TEXT
GLOBAL _start

_start:
        mov eax, 4        ; write(
        mov ebx, 1        ;   STDOUT_FILENO,
        mov ecx, Msg      ;   "Hello, world!\n",
        mov edx, MsgLen   ;   sizeof("Hello, world!\n")
        int 80h           ; );

        mov eax, 1        ; exit(
        mov ebx, 0        ;   EXIT_SUCCESS
        int 80h           ; );
```

System call numbers:

```
$ less /usr/include/asm/unistd_32.h
$ less /usr/include/asm/unistd_64.h
```

System Call Examples

```c
#include <unistd.h>

int main(void)
{
  write(1, "Hello, world!\n", 14);

  return 0;
}
```

- Actually, write() is a wrapper function in glibc.

$ man 2 write

$ man 3 write

Don't invoke syscall directly whenever possible

```c
int main(void) {
  register char* arg2 asm("rsi") = "hello, world!\n";

  /* rax: sys_write; rdi: STDOUT; */
  asm("mov $1, %rax; mov $1, %rdi; mov $14, %rdx; syscall;");

  return 0;
}
```

System Call Examples

```
$ man 2 fork
```

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

int main ()
{
  puts("Hello World!");
  fork();
  puts("Goodbye Cruel World!");
  return 0;
}
```

```
$ man 3 exec
```

```c
int main()
{
  printf("Main process (%d) saying hello!\n", getpid());

  if(fork() == 0 ){                /* child */
    printf("Child (%d) is listing the source file...\n", getpid());

    execl("/bin/ls", "", NULL);

    /* useless after execl() */
    puts("You can't see this line unless execl() failed.\n");
  }
  else {                           /* parent */
    int i=60;
    printf("parent (%d) is sleeping for %d seconds...\n", getpid(),i);
    sleep(i);                      /* for zombie */
  }

  /* only the one who doesn't call execl() can do the following. */
  printf("Hello again from process %d\n", getpid());

  return 0;
}
```

Hardware INT vs. Software INT

Device:
Send electrical signal to interrupt controller.

Controller:
1. Interrupt CPU.
2. Send digital identification of interrupting device.

Kernel:
1. Save registers.
2. Execute driver software to read I/O device.
3. Send message.
4. Restart a process (not necessarily interrupted process).

(a)

Caller:
1. Put message pointer and destination of message into CPU registers.
2. Execute software interrupt instruction.

Kernel:
1. Save registers.
2. Send and/or receive message.
3. Restart a process (not necessarily calling process).

(b)

References

Wikipedia. *Interrupt — Wikipedia, The Free Encyclopedia*. 2015. http://en.wikipedia.org/w/index.php?title=Interrupt&oldid=646521061.

Wikipedia. *System call — Wikipedia, The Free Encyclopedia*. 2015. http://en.wikipedia.org/w/index.php?title=System%5C_call&oldid=647910319.

Part II

Process And Thread

7 Processes

7.1 What's a Process

Process

A process is an instance of a program in execution

Processes are like human beings:

- they are generated
- they have a life
- they optionally generate one or more child processes, and
- eventually they die

A small difference:

- sex is not really common among processes
- each process has just one parent

7.2 PCB

Process Control Block (PCB)

Implementation

A process is the collection of data structures that fully describes how far the execution of the program has progressed.

- Each process is represented by a PCB
- `task_struct` in 🐧

`process state`
`PID`
`program counter`
`registers`
`memory limits`
`list of open files`
`...`

7.3 Process Creation

Process Creation

- When a process is created, it is almost identical to its parent
  - It receives a (logical) copy of the parent's address space, and
  - executes the same code as the parent
- The parent and child have separate copies of the data (stack and heap)

Forking in C

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

int main ()
{
  puts("Hello World!");
  fork();
  puts("Goodbye Cruel World!");
  return 0;
}
```

```
$ man fork
```

exec()

```
int main()
{
  pid_t pid;

  pid = fork();
  if(pid < 0) {          /* error */
    fprintf(stderr, "Fork Failed");
    exit(-1);
  }
  else if(pid == 0){     /* child */
    execlp("/bin/ls", "ls", NULL);
  }
  else {                 /* parent */
    wait(NULL);
    puts("Child Complete");
    exit(EXIT_SUCCESS);
  }
  return 0;
}
```

```
$ man 3 exec
```

7.4 Process State

Process State Transition

1. Process blocks for input
2. Scheduler picks another process
3. Scheduler picks this process
4. Input becomes available

7.5 CPU Switch From Process To Process

CPU Switch From Process To Process

8 Threads

8.1 Processes vs. Threads

Process vs. Thread

a single-threaded process = resource + execution
a multi-threaded process = resource + executions

A process = a unit of resource ownership, used to group resources together;

A thread = a unit of scheduling, scheduled for execution on the CPU.

Process vs. Thread

multiple threads running in one process:	**multiple processes running in one computer:**
share an address space and other resources	share physical memory, disk, printers ...

No protection between threads

impossible — because process is the minimum unit of resource management

unnecessary — a process is owned by a single user

Threads

code, data, open files, signals...		
thread ID	thread ID	thread ID
program counter	program counter	program counter
register set	register set	register set
stack	stack	stack

8.2 Why Thread?

A Multi-threaded Web Server

```
while (TRUE) {
    get_next_request(&buf);
    handoff_work(&buf);
}
```

(a)

```
while (TRUE) {
    wait_for_work(&buf)
    look_for_page_in_cache(&buf, &page);
    if (page_not_in_cache(&page))
        read_page_from_disk(&buf, &page);
    return_page(&page);
}
```

(b)

A Word Processor With 3 Threads

Why Having a Kind of Process Within a Process?

- Responsiveness
  - Good for interactive applications.
  - A process with multiple threads makes a great server (e.g. a web server):

    Have one server process, many ”worker” threads – if one thread blocks (e.g. on a read), others can still continue executing
- Economy – Threads are cheap!
  - Cheap to create – only need a stack and storage for registers
  - Use very little resources – don’t need new address space, global data, program code, or OS resources
  - switches are fast – only have to save/restore PC, SP, and registers
- Resource sharing – Threads can pass data via shared memory; no need for IPC
- Can take advantage of multiprocessors

8.3 Thread Characteristics

Thread States Transition

Same as process states transition

1. Process blocks for input
2. Scheduler picks another process
3. Scheduler picks this process
4. Input becomes available

Each Thread Has Its Own Stack

- A typical stack stores local data and call information for (usually nested) procedure calls.
- Each thread generally has a different execution history.

Thread Operations

8.4 POSIX Threads

POSIX Threads

IEEE 1003.1c The standard for writing portable threaded programs. The threads package it defines is called Pthreads, including over 60 function calls, supported by most UNIX systems.

Some of the Pthreads function calls

Thread call	Description
`pthread_create`	Create a new thread
`pthread_exit`	Terminate the calling thread
`pthread_join`	Wait for a specific thread to exit
`pthread_yield`	Release the CPU to let another thread run
`pthread_attr_init`	Create and initialize a thread's attribute structure
`pthread_attr_destroy`	Remove a thread's attribute structure

Pthreads

Example 1

```
void *hello(void *arg)
{
  int i;
  for( i=0; i<20; i++ ){
    puts("Thread says hi!");
    sleep(1);
  }
  return NULL;
}

int main(void)
{
  pthread_t t;
  if( pthread_create(&t, NULL, hello, NULL) ){}

  if( pthread_join(t, NULL) ){}
  exit(0);
}
```

Pthreads

`pthread_t` defined in `pthread.h`, is often called a ”thread id” (`tid`);

`pthread_create()` returns zero on success and a non-zero value on failure;

`pthread_join()` returns zero on success and a non-zero value on failure;

How to use pthread?

```
  #include<pthread.h>
$ gcc thread1.c -o thread1 -pthread
$ ./thread1
```

Pthreads

Example 2

```
#define NUMBER_OF_THREADS 5

void *hello(void *tid)
{
  printf ("Hello from thread %d\n", *(int*)tid);
  pthread_exit(NULL);
}

int main(void)
{
  pthread_t t[NUMBER_OF_THREADS];
  int status, i;


  for (i=0; i<NUMBER_OF_THREADS; i++){
    printf("Main: creating thread %d ...", i);

    if( (status = pthread_create(&t[i], NULL, hello, (void *)&i)) ){}
    puts("done.");
  }

  for (i=0; i<NUMBER_OF_THREADS; i++){
    printf("Joining thread %d ...",i);

    if( pthread_join(t[i], NULL) ){}
    puts("done.");
  }
  exit(0);
}
```

Pthreads

With or without `pthread_join()`? Check it by yourself.

8.5 User-Level Threads vs. Kernel-level Threads

User-Level Threads vs. Kernel-Level Threads

User-Level Threads

User-level threads provide a library of functions to allow user processes to create and manage their own threads.

- ☺ No need to modify the OS
- ☺ Simple representation
  - each thread is represented simply by a PC, regs, stack, and a small TCB, all stored in the user process' address space
- ☺ Simple Management
  - creating a new thread, switching between threads, and synchronization between threads can all be done without intervention of the kernel
- ☺ Fast
  - thread switching is not much more expensive than a procedure call
- ☺ Flexible
  - CPU scheduling (among threads) can be customized to suit the needs of the algorithm – each process can use a different thread scheduling algorithm

User-Level Threads

- ☹ Lack of coordination between threads and OS kernel
  - Process as a whole gets one time slice
  - Same time slice, whether process has 1 thread or 1000 threads
  - Also – up to each thread to relinquish control to other threads in that process
- ☹ Requires non-blocking system calls (i.e. a multithreaded kernel)
  - Otherwise, entire process will blocked in the kernel, even if there are runnable threads left in the process
  - part of motivation for user-level threads was not to have to modify the OS
- ☹ If one thread causes a page fault(interrupt!), the entire process blocks

Kernel-Level Threads

Kernel-level threads kernel provides system calls to create and manage threads

- ☺ Kernel has full knowledge of all threads
  - Scheduler may choose to give a process with 10 threads more time than process with only 1 thread
- ☺ Good for applications that frequently block (e.g. server processes with frequent interprocess communication)
- ☹ Slow – thread operations are 100s of times slower than for user-level threads
- ☹ Significant overhead and increased kernel complexity – kernel must manage and schedule threads as well as processes
  - Requires a full thread control block (TCB) for each thread

Hybrid Implementations

Combine the advantages of two

Programming Complications

- `fork()`: shall the child has the threads that its parent has?
- What happens if one thread closes a file while another is still reading from it?
- What happens if several threads notice that there is too little memory?

And sometimes, threads fix the symptom, but not the problem.

8.6 Linux Threads

Linux Threads

To the Linux kernel, there is no concept of a thread

- Linux implements all threads as standard processes
- To Linux, a thread is merely a process that shares certain resources with other processes
- Some OS (MS Windows, Sun Solaris) have cheap threads and expensive processes.
- Linux processes are already quite lightweight

On a 75MHz Pentium
thread: $1.7\mu s$
fork: $1.8\mu s$

Linux Threads

`clone()` creates a separate process that shares the address space of the calling process. The cloned task behaves *much like* a separate thread.

clone()

```
1  #include <sched.h>
2  int clone(int (*fn) (void *), void *child_stack,
3            int flags, void *arg, ...);
```

- **arg 1** the function to be executed, i.e. `fn(arg)`, which returns an `int`;
- **arg 2** a pointer ☛ a (usually malloced) memory space to be used as the stack for the new thread;
- **arg 3** a set of flags used to indicate how much the calling process is to be shared. In fact,

  `clone(0) == fork()`

- **arg 4** the arguments passed to the function.

It returns the PID of the child process or -1 on failure.

```
$ man clone
```

The `clone()` System Call

Some flags:

flag	Shared
`CLONE_FS`	File-system info
`CLONE_VM`	Same memory space
`CLONE_SIGHAND`	Signal handlers
`CLONE_FILES`	The set of open files

In practice, one should try to avoid calling `clone()` directly

Instead, use a threading library (such as pthreads) which use `clone()` when starting a thread (such as during a call to `pthread_create()`)

clone() Example

```c
#include <unistd.h>
#include <sched.h>
#include <sys/types.h>
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <fcntl.h>

int variable;

int do_something()
{
  variable = 42;
  _exit(0);
}

int main(void)
{
  void *child_stack;
  variable = 9;

  child_stack = (void *) malloc(8192);
  printf("The variable was %d\n", variable);

  clone(do_something, child_stack,
        CLONE_FS | CLONE_VM | CLONE_FILES, NULL);
  sleep(1);

  printf("The variable is now %d\n", variable);
  return 0;
}
```

clone() Example

```c
#include <unistd.h>
#include <sched.h>
#include <sys/types.h>
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <fcntl.h>

int variable;

int do_something()
{
  variable = 42;
  _exit(0);
}
```

```c
int main(void)
{
  void *child_stack;
  variable = 9;

  child_stack = (void *) malloc(8192);
  printf("The variable was %d\n", variable);

  clone(do_something, child_stack,
        CLONE_FS | CLONE_VM | CLONE_FILES, NULL);
  sleep(1);

  printf("The variable is now %d\n", variable);
  return 0;
}
```

Stack grows downwards

```c
child_stack = (void**)malloc(8192) + 8192/sizeof(*child_stack);
```

References

Wikipedia. *Process (computing) — Wikipedia, The Free Encyclopedia*. 2014. http://en.wikipedia.org/w/index.php?title=Process%5C_(computing)&oldid=639847817.

Wikipedia. *Thread (computing) — Wikipedia, The Free Encyclopedia*. 2015. http://en.wikipedia.org/w/index.php?title=Thread%5C_(computing)&oldid=648172980.

Wikipedia. *Process control block — Wikipedia, The Free Encyclopedia*. 2015. http://en.wikipedia.org/w/index.php?title=Process%5C_control%5C_block&oldid=646933587.

9 Process Synchronization

9.1 IPC

Interprocess Communication

Example:

```
$ unicode skull | head -1 | cut -f1 -d' ' | sm -
```

IPC issues:

1. How one process can pass information to another
2. Be sure processes do not get into each other's way
   e.g. in an airline reservation system, two processes compete for the last seat
3. Proper sequencing when dependencies are present
   e.g. if A produces data and B prints them, B has to wait until A has produced some data

Two models of IPC:

- Shared memory
- Message passing (e.g. sockets)

Producer-Consumer Problem

9.2 Shared Memory

Process Synchronization

Producer-Consumer Problem

- Consumers don't try to remove objects from Buffer when it is empty.
- Producers don't try to add objects to the Buffer when it is full.

```
while(TRUE){
    while(FULL);
    item = produceItem();
    insertItem(item);
}
```

```
while(TRUE){
    while(EMPTY);
    item = removeItem();
    consumeItem(item);
}
```

How to define full/empty?

Producer-Consumer Problem

— Bounded-Buffer Problem (Circular Array)

Front(out): the first full position

Rear(in): the next free position

Full or empty when "*front* $==$ *rear*"?

Producer-Consumer Problem

Common solution:

Full: when "$(\texttt{in} + 1)\%\texttt{BUFFER_SIZE} == \texttt{out}$"

Actually, this is "$\texttt{full} - 1$"

Empty: when "$\texttt{in} == \texttt{out}$"

Can only use "$\texttt{BUFFER_SIZE} - 1$" elements

Shared data:

```
#define BUFFER_SIZE 6
typedef struct {
        /* ... */
} item;
item buffer[BUFFER_SIZE];
int in = 0; //the next free position
int out = 0;//the first full position
```

Bounded-Buffer Problem

Producer:

```
while (true) {
  /* do nothing -- no free buffers */
  while (((in + 1) % BUFFER_SIZE) == out);

  produce(buffer[in]);

  in = (in + 1) % BUFFER_SIZE;
}
```

Consumer:

```
while (true) {
  while (in == out); // do nothing

  consume(buffer[out]);

  out = (out + 1) % BUFFER_SIZE;
}
```

Pipe

```
$ ls | wc -l
```

- A pipe is a byte stream
- Unidirectional
- `read()` would be blocked if nothing written at the other end

tee

```
$ ls | tee ls.out
```

- No direct link between A and B (need system calls)
- A pipe is simply a buffer maintained in kernel memory

```
$ cat /proc/sys/fs/pipe-max-size
```

pipe()

```
1  #include <unistd.h>
2
3  int pipe(int fd[2]);
```

pipe() + fork()

a) After *fork()*

b) After closing unused descriptors

```c
#define BUF_SIZE 10

int main(int argc, char *argv[]) /* Over-simplified! */
{
  int pfd[2]; /* Pipe file descriptors */
  char buf[BUF_SIZE];
  ssize_t numRead;

  pipe(pfd); /* Create the pipe */

  switch (fork()) {
  case 0: /* Child - reads from pipe */
    close(pfd[1]); /* Write end is unused */

    for(;;) { /* Read data from pipe, echo on stdout */
      if( (numRead = read(pfd[0], buf, BUF_SIZE)) == 0 )
        break;  /* End-of-file */
      if( write(1, buf, numRead) != numRead ){}
    }
    puts(""); /* newline */

    close(pfd[0]); _exit(EXIT_SUCCESS);

  default: /* Parent - writes to pipe */
    close(pfd[0]); /* Read end is unused */

    if( (size_t)write(pfd[1], argv[1], strlen(argv[1])) != strlen(argv[1]) ){}

    close(pfd[1]); /* Child will see EOF */

    wait(NULL); /* Wait for child to finish */
    exit(EXIT_SUCCESS);
  }
}
```

Named Pipe (FIFO)

PIPEs pass data between related processes.

FIFOs pass data between any processes.

```
$ mkfifo myfifo
  $ echo hello > myfifo
  $ cat myfifo
```

tee

```
  $ echo hello | tee myfifo
  $ wc myfifo
```

IPC With FIFO

```
#define FIFO_NAME "/tmp/myfifo"

int main(int argc, char *argv[]) /* Oversimplified */
{
  int fd, i, mode = 0;
  char c;

  if (argc < 2) {}

  for(i = 1; i < argc; i++) {
    if (strncmp(*++argv, "O_RDONLY", 8) == 0)  mode |= O_RDONLY;
    if (strncmp(*argv, "O_WRONLY", 8) == 0)    mode |= O_WRONLY;
    if (strncmp(*argv, "O_NONBLOCK", 10) == 0) mode |= O_NONBLOCK;
  }

  if (access(FIFO_NAME, F_OK) == -1) mkfifo(FIFO_NAME, 0777);

  printf("Process %d: FIFO(fd %d, mode %d) opened.\n",
         getpid(), fd = open(FIFO_NAME, mode), mode);

  if( (mode == 0) | (mode == 2048) )
      while( read(fd,&c,1) == 1 ) putchar(c);

  if( (mode == 1) | (mode == 2049) )
      while( (c = getchar()) != EOF ) write(fd,&c,1);

  exit(EXIT_SUCCESS);
}
```

```
$ watch 'lsof -n.1 /tmp/myfifo'
$ ./a.out O_RDONLY
$ ./a.out O_WRONLY
$ ./a.out O_RDONLY O_NONBLOCK
$ ./a.out O_WRONLY O_NONBLOCK
```

O_NONBLOCK

- A `read()/write()` will wait on an empty blocking FIFO
- A `read()` on an empty nonblocking FIFO will return 0 bytes
- `open(const char *path, O_WRONLY | O_NONBLOCK);`
  - Returns an error (-1) if FIFO not open
  - Okay if someone's reading the FIFO
- If opened with `O_RDWR`, the result is undefined

Message Queues

Message Queues

Send

```
int main(int argc, char **argv)
{
  mqd_t queue;
  struct mq_attr attrs;
  size_t msg_len;

  if (argc < 3){}

  queue = mq_open(argv[1], O_WRONLY | O_CREAT, S_IRUSR | S_IWUSR, NULL);
  if (queue == (mqd_t)-1){}

  if (mq_getattr(queue, &attrs) == -1){}

  msg_len = strlen(argv[2]);
  if (msg_len > LONG_MAX || (long)msg_len > attrs.mq_msgsize){}

  if (mq_send(queue, argv[2], strlen(argv[2]), 0) == -1){}

  return 0;
}
```

Message Queues

Receive

```
int main(int argc, char **argv)
{
  mqd_t queue;
  struct mq_attr attrs;
  char *msg_ptr;
  ssize_t recvd;
  size_t i;

  if (argc < 2){}

  queue = mq_open(argv[1], O_RDONLY | O_CREAT, S_IRUSR | S_IWUSR, NULL);
  if (queue == (mqd_t)-1){}

  if (mq_getattr(queue, &attrs) == -1){}

  msg_ptr = calloc(1, attrs.mq_msgsize);
  if (msg_ptr == NULL){}

  recvd = mq_receive(queue, msg_ptr, attrs.mq_msgsize, NULL);
  if (recvd == -1){}

  printf("Message: ");
  for (i = 0; i < (size_t)recvd; i++)
    putchar(msg_ptr[i]);
  puts("");
}
```

Relationship Between Kernel Data Structures

9.3 Race Condition and Mutual Exclusion

Race Conditions

Now, let's have two producers

Race Conditions

Two producers

```c
#define BUFFER_SIZE 100
typedef struct {
        /* ... */
} item;
item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

Process A and B do the same thing:

```c
while (true) {
  while (((in + 1) % BUFFER_SIZE) == out);
  buffer[in] = item;
  in = (in + 1) % BUFFER_SIZE;
}
```

Race Conditions

Problem: Process `B` started using one of the *shared variables* before Process `A` was finished with it.

Solution: Mutual exclusion. If one process is using a shared variable or file, the other processes will be excluded from doing the same thing.

Critical Regions

Mutual Exclusion

Critical Region: is a piece of code accessing a common resource.

Critical Region

A solution to the critical region problem must satisfy three conditions:

Mutual Exclusion: No two process may be simultaneously inside their critical regions.

Progress: No process running outside its critical region may block other processes.

Bounded Waiting: No process should have to wait forever to enter its critical region.

Mutual Exclusion With Busy Waiting

Disabling Interrupts

```
{
  ...
  disableINT();
  critical_code();
  enableINT();
  ...
}
```

Problems:

- It's not wise to give user process the power of turning off INTs.
  - Suppose one did it, and never turned them on again
- useless for multiprocessor system

Disabling INTs is often a useful technique within the kernel itself but is not a general mutual exclusion mechanism for user processes.

Mutual Exclusion With Busy Waiting

Lock Variables

```
1  int lock=0; //shared variable
2  {
3    ...
4    while(lock); //busy waiting
5    lock=1;
6    critical_code();
7    lock=0;
8    ...
9  }
```

Problem:

- What if an interrupt occurs right at line 5?
- Checking the lock again while backing from an interrupt?

Mutual Exclusion With Busy Waiting

Strict Alternation

```
while(TRUE){
  while(turn != 0);
  critical_region();
  turn = 1;
  noncritical_region();
}
```

```
while(TRUE){
  while(turn != 1);
  critical_region();
  turn = 0;
  noncritical_region();
}
```

Problem: violates condition-2

- One process can be blocked by another not in its critical region.
- Requires the two processes strictly alternate in entering their critical region.

Mutual Exclusion With Busy Waiting

Peterson's Solution

```
int interest[0] = 0;
int interest[1] = 0;
int turn;
```

P0

```
interest[0] = 1;
turn = 1;
while(interest[1] == 1
          && turn == 1);
critical_section();
interest[0] = 0;
```

P1

```
interest[1] = 1;
turn = 0;
while(interest[0] == 1
          && turn == 0);
critical_section();
interest[1] = 0;
```

Wikipedia. *Peterson's algorithm — Wikipedia, The Free Encyclopedia*. 2015.

Mutual Exclusion With Busy Waiting

Lock file

```
const char *mylock = "/tmp/LCK.test2";

int main() {
  int fd;

  for(;;){
    while( (fd = open(mylock, O_RDWR | O_CREAT | O_EXCL, 0444)) != -1 ){
      printf("Process(%d) - Working in critical region...\n", getpid());
      sleep(2);            /* working */
      close(fd);
      if ( unlink(mylock) == 0 ) puts("Done.\nResource unlocked.");
      sleep(3);            /* non-critical region */
    }
    printf("Process(%d) - Waiting for lock...\n", getpid());
  }
  exit(EXIT_SUCCESS);
}
```

☹ Lock file could be left in system after Ctrl + c

✘

```
const char *mylock = "/tmp/LCK.test2";

void sigint(int signo){
  if ( unlink(mylock) == 0 ) puts("Quit. Lock released.");
  exit(EXIT_SUCCESS);
}

int main() {
  int fd;

  signal(SIGINT,sigint);

  for(;;){
    while( (fd = open(mylock, O_RDWR | O_CREAT | O_EXCL, 0444)) != -1 ) {
      printf("Process(%d) - Working in critical region...\n", getpid());
      sleep(2);            /* working */
      close(fd);
      if ( unlink(mylock) == 0 ) puts("Done.\nResource unlocked.");
      sleep(3);            /* non-critical region */
    }
    printf("Process(%d) - Waiting for lock...\n", getpid());
  }
  exit(EXIT_SUCCESS);
}
```

Mutual Exclusion With Busy Waiting

Hardware Solution: The TSL Instruction

Lock the memory bus

```
enter_region:
    TSL REGISTER,LOCK        | copy lock to register and set lock to 1
    CMP REGISTER,#0          | was lock zero?
    JNE enter_region         | if it was non zero, lock was set, so loop
    RET                      | return to caller; critical region entered

leave_region:
    MOVE LOCK,#0             | store a 0 in lock
    RET                      | return to caller
```

Mutual Exclusion Without Busy Waiting

Sleep & Wakeup

```c
#define N 100  /* number of slots in the buffer */
int count = 0; /* number of items in the buffer */
```

```c
void producer(){
  int item;
  while(TRUE){
    item = produce_item();
    if(count == N)
      sleep();
    insert_item(item);
    count++;
    if(count == 1)
      wakeup(consumer);
  }
}
```

```c
void consumer(){
  int item;
  while(TRUE){
    if(count == 0)
      sleep();
    item = rm_item();
    count--;
    if(count == N - 1)
      wakeup(producer);
    consume_item(item);
  }
}
```

Mutual Exclusion Without Busy Waiting

Sleep & Wakeup

```c
#define N 100  /* number of slots in the buffer */
int count = 0; /* number of items in the buffer */
```

```c
void producer(){
  int item;
  while(TRUE){
    item = produce_item();
    if(count == N)
      sleep();
    insert_item(item);
    count++;
    if(count == 1)
      wakeup(consumer);
  }
}
```

```c
void consumer(){
  int item;
  while(TRUE){
    if(count == 0)
      sleep();
    item = rm_item();
    count--;
    if(count == N - 1)
      wakeup(producer);
    consume_item(item);
  }
}
```

Deadlock!

Producer-Consumer Problem

Race Condition

Problem

1. Consumer is going to sleep upon seeing an empty buffer, but INT occurs;
2. Producer inserts an item, increasing `count` to `1`, then call `wakeup(consumer)`;
3. But the consumer is not asleep, though `count` was `0`. So *the `wakeup()` signal is lost*;
4. Consumer is back from INT remembering `count` is `0`, and goes to sleep;
5. Producer sooner or later will fill up the buffer and also goes to sleep;
6. Both will sleep forever, and waiting to be waken up by the other process. Deadlock!

Producer-Consumer Problem

Race Condition

Solution: Add a *wakeup waiting bit*

1. The bit is set, when a wakeup is sent to an awaken process;
2. Later, when the process wants to sleep, it checks the bit first. Turns it off if it's set, and stays awake.

What if many processes try going to sleep?

9.4 Semaphores

What is a Semaphore?

- A locking mechanism
- An integer or ADT

Atomic Operations	
P()	V()
Wait()	Signal()
Down()	Up()
Decrement()	Increment()
...	...

```
down(S){
  while(S<=0);
  S--;
}
```

```
up(S){
  S++;
}
```

More meaningful names:

- `increment_and_wake_a_waiting_process_if_any()`
- `decrement_and_block_if_the_result_is_negative()`

Semaphore

How to ensure atomic?

1. For single CPU, implement `up()` and `down()` as system calls, with the OS disabling all interrupts while accessing the semaphore;
2. For multiple CPUs, to make sure only one CPU at a time examines the semaphore, a lock variable should be used with the TSL instructions.

Semaphore is a Special Integer

A semaphore is like an integer, with three differences:

1. You can initialize its value to any integer, but after that the only operations you are allowed to perform are *increment* (`S++`) and *decrement* (`S--`).
2. When a thread decrements the semaphore, if the result is negative ($S \leq 0$), the thread blocks itself and cannot continue until another thread increments the semaphore.
3. When a thread increments the semaphore, if there are other threads waiting, one of the waiting threads gets unblocked.

Why Semaphores?

We don't need semaphores to solve synchronization problems, but there are some advantages to using them:

- Semaphores impose deliberate constraints that help programmers avoid errors.
- Solutions using semaphores are often clean and organized, making it easy to demonstrate their correctness.
- Semaphores can be implemented efficiently on many systems, so solutions that use semaphores are portable and usually efficient.

The Simplest Use of Semaphore

Signaling

- One thread sends a signal to another thread to indicate that something has happened
- it solves the serialization problem

  Signaling makes it possible to guarantee that a section of code in one thread will run before a section of code in another thread

```
statement a1
sem.signal()
```

```
sem.wait()
statement b1
```

What's the initial value of `sem`?

Semaphore

Rendezvous Puzzle

```
statement a1
statement a2
```

```
statement b1
statement b2
```

Q: How to guarantee that

1. `a1` happens before `b2`, and
2. `b1` happens before `a2`

$\texttt{a1} \rightarrow \texttt{b2}; \quad \texttt{b1} \rightarrow \texttt{a2}$

Hint: Use two semaphores initialized to 0.

	Thread A:	Thread B:

Solution 1:	`statement a1`
	`sem1.wait()`
	`sem2.signal()`
	`statement a2`
		`statement b1`
		`sem1.signal()`
		`sem2.wait()`
		`statement b2`

Solution 2:	`statement a1`
	`sem2.signal()`
	`sem1.wait()`
	`statement a2`
		`statement b1`
		`sem1.signal()`
		`sem2.wait()`
		`statement b2`

Solution 3:	`statement a1`
	`sem2.wait()`
	`sem1.signal()`
	`statement a2`
		`statement b1`
		`sem1.wait()`
		`sem2.signal()`
		`statement b2`

	Thread A:	Thread B:

Solution 1:	`statement a1`
	`sem1.wait()`
	`sem2.signal()`
	`statement a2`
		`statement b1`
		`sem1.signal()`
		`sem2.wait()`
		`statement b2`

Solution 2:	`statement a1`
	`sem2.signal()`
	`sem1.wait()`
	`statement a2`
		`statement b1`
		`sem1.signal()`
		`sem2.wait()`
		`statement b2`

Solution 3:	`statement a1`
	`sem2.wait()`
	`sem1.signal()`
	`statement a2`
		`statement b1`
		`sem1.wait()`
		`sem2.signal()`
		`statement b2`

Deadlock!

Example

```
void *func(void *arg);
sem_t sem;
#define BUFSIZE 1024
char buf[BUFSIZE];

int main() {
  pthread_t t;

  if( sem_init(&sem, 0, 0) != 0 ) {}

  if( pthread_create(&t, NULL, func, NULL) != 0 ) {}

  puts("Please input some text. Ctrl-d to quit.");

  while( fgets(buf, BUFSIZE, stdin) )
    sem_post(&sem);

  sem_post(&sem);       /* in case of Ctrl-d */

  if( pthread_join(t, NULL) != 0) {}

  sem_destroy(&sem);  exit(EXIT_SUCCESS);
}

void *func(void *arg) {
  sem_wait(&sem);
  while( buf[0] != '\0' ) {
    printf("You input %ld characters\n", strlen(buf)-1);
    buf[0] = '\0';      /* in case of Ctrl-d */
    sem_wait(&sem);
  }
  pthread_exit(NULL);
}
```

Mutex

- A second common use for semaphores is to enforce mutual exclusion
- It guarantees that only one thread accesses the shared variable at a time
- A `mutex` is like a token that passes from one thread to another, allowing one thread at a time to proceed

Q: Add semaphores to the following example to enforce mutual exclusion to the shared variable `i`.

Thread A: `i++`

Thread B: `i++`

Why? Because `i++` is not atomic.

i++ can go wrong!

```
static int glob = 0;

static void *threadFunc(void *arg) /* loop 'arg' times */
{
  int j;
  for (j = 0; j < *((int *) arg); j++) glob++; /* not atomic! */
  return NULL;
}

int main(int argc, char *argv[])
{
  pthread_t t1, t2;
  int loops;

  loops = (argc > 1) ? atoi(argv[1]) : 10000000;

  if( pthread_create(&t1, NULL, threadFunc, &loops) != 0 ){}

  if( pthread_create(&t2, NULL, threadFunc, &loops) != 0 ){}

  if( pthread_join(t1, NULL) != 0 ){}

  if( pthread_join(t2, NULL) != 0 ){}

  printf("glob = %d\n", glob);
  exit(EXIT_SUCCESS);
}
```

`i++` is not atomic in assembly language

```
LOAD    [i], r0    ;load the value of 'i' into
                   ;a register from memory
ADD     r0, 1      ;increment the value
                   ;in the register
STORE   r0, [i]    ;write the updated
                   ;value back to memory
```

Interrupts might occur in between. So, `i++` needs to be protected with a `mutex`.

Mutex Solution

Create a semaphore named `mutex` that is initialized to `1`

1: a thread may proceed and access the shared variable

0: it has to wait for another thread to release the mutex

```
mutex.wait()
    i++
mutex.signal()
```

```
mutex.wait()
    i++
mutex.signal()
```

```c
static int glob = 0;
static pthread_mutex_t mtx = PTHREAD_MUTEX_INITIALIZER;

static void *threadFunc(void *arg)
{
  int j;
  for (j = 0; j < *((int *) arg); j++) {
    if ( pthread_mutex_lock(&mtx) != 0 ){}
    glob++;
    if ( pthread_mutex_unlock(&mtx) != 0){}
  }
  return NULL;
}

int main(int argc, char *argv[])
{
  pthread_t t1, t2;
  int loops;

  loops = (argc > 1) ? atoi(argv[1]) : 10000000;

  if( pthread_create(&t1, NULL, threadFunc, &loops) != 0 ){}

  if( pthread_create(&t2, NULL, threadFunc, &loops) != 0 ){}

  if( pthread_join(t1, NULL) != 0 ){}

  if( pthread_join(t2, NULL) != 0 ){}

  printf("glob = %d\n", glob);
  exit(EXIT_SUCCESS);
}
```

```
static int glob = 0;
static sem_t sem;

static void *threadFunc(void *arg)
{
  int j;
  for (j = 0; j < *((int *) arg); j++) {
    if (sem_wait(&sem) == -1){}
    glob++;
    if (sem_post(&sem) == -1){}
  }
  return NULL;
}
```

Multiplex — Without Busy Waiting

```c
typedef struct{
  int space;          //number of free resources
  struct process *P; //a list of queueing producers
  struct process *C; //a list of queueing consumers
} semaphore;
semaphore S;
S.space = 5;
```

```c
void down(S){
  S.space--;
  if(S.space == 4){
    rmFromQueue(S.C);
    wakeup(S.C);
  }
  if(S.space < 0){
    addToQueue(S.P);
    sleep();
  }
}
```

```c
void up(S){
  S.space++;
  if(S.space > 5){
    addToQueue(S.C);
    sleep();
  }
  if(S.space >= 0){
    rmFromQueue(S.P);
    wakeup(S.P);
  }
}
```

if `S.space < 0`,
`S.space == Number of queuing producers`

if `S.space > 5`,
`S.space == Number of queuing consumers + 5`

Barrier

1. Processes approaching a barrier
2. All processes but one blocked at the barrier
3. When the last process arrives at the barrier, all of them are let through

Synchronization requirement:

```
specific_task()
critical_point()
```

No thread executes `critical_point()` until after all threads have executed `specific_task()`.

Barrier Solution

```
n = the number of threads
count = 0
mutex = Semaphore(1)
barrier = Semaphore(0)
```

count: keeps track of how many threads have arrived

mutex: provides exclusive access to `count`

barrier: is locked (≤ 0) until all threads arrive

When `barrier.value<0`,

`barrier.value == Number of queueing processes`

```
specific_task();
mutex.wait();
   count++;
mutex.signal();
if (count < n)
   barrier.wait();
barrier.signal();
critical_point();
```

```
specific_task();
mutex.wait();
   count++;
mutex.signal();
if (count == n)
   barrier.signal();
barrier.wait();
critical_point();
```

Barrier Solution

```
n = the number of threads
count = 0
mutex = Semaphore(1)
barrier = Semaphore(0)
```

count: keeps track of how many threads have arrived

mutex: provides exclusive access to `count`

barrier: is locked (≤ 0) until all threads arrive

When `barrier.value<0`,

`barrier.value == Number of queueing processes`

```
specific_task();
mutex.wait();
    count++;
mutex.signal();
if (count < n)
    barrier.wait();
barrier.signal();
critical_point();
```

```
specific_task();
mutex.wait();
    count++;
mutex.signal();
if (count == n)
    barrier.signal();
barrier.wait();
critical_point();
```

Deadlock!

Only one thread can pass the barrier!

Barrier Solution

```
specific_task();

mutex.wait();
   count++;
mutex.signal();

if (count == n)
   barrier.signal();

barrier.wait();
barrier.signal();

critical_point();
```

```
specific_task();

mutex.wait();
   count++;

   if (count == n)
      barrier.signal();

   barrier.wait();
   barrier.signal();
mutex.signal();

critical_point();
```

Barrier Solution

```
specific_task();

mutex.wait();
    count++;
mutex.signal();

if (count == n)
    barrier.signal();

barrier.wait();
barrier.signal();

critical_point();
```

```
specific_task();

mutex.wait();
    count++;

    if (count == n)
        barrier.signal();

    barrier.wait();
    barrier.signal();
mutex.signal();

critical_point();
```

Deadlock!

☠ Blocking on a semaphore while holding a mutex! ☠

```
barrier.wait();
barrier.signal();
```

Turnstile

This pattern, a `wait` and a `signal` in rapid succession, occurs often enough that it has a name called a *turnstile*, because

- it allows one thread to pass at a time, and
- it can be locked to bar all threads

Semaphores

Producer-Consumer Problem

Whenever an event occurs

- a producer thread creates an *event object* and adds it to the event buffer. Concurrently,
- consumer threads take events out of the buffer and process them. In this case, the consumers are called "*event handlers*".

```
event = waitForEvent()
buffer.add(event)
```

```
event = buffer.get()
event.process()
```

Q: Add synchronization statements to the producer and consumer code to enforce the synchronization constraints

1. Mutual exclusion
2. Serialization

Semaphores — Producer-Consumer Problem

Solution

Initialization:

```
mutex = Semaphore(1)
items = Semaphore(0)
```

- `mutex` provides exclusive access to the buffer
- `items`:
  - ➕ number of items in the buffer
  - ➖ number of consumer threads in queue

Semaphores — Producer-Consumer Problem

Solution

```
event = waitForEvent()
mutex.wait()
    buffer.add(event)
    items.signal()
mutex.signal()
```

```
items.wait()
mutex.wait()
    event = buffer.get()
mutex.signal()
event.process()
```

or,

```
event = waitForEvent()
mutex.wait()
    buffer.add(event)
mutex.signal()
items.signal()
```

or,

```
mutex.wait()
    items.wait()
    event = buffer.get()
mutex.signal()
event.process()
```

Semaphores — Producer-Consumer Problem

Solution

```
event = waitForEvent()
mutex.wait()
    buffer.add(event)
    items.signal()
mutex.signal()
```

```
items.wait()
mutex.wait()
    event = buffer.get()
mutex.signal()
event.process()
```

or,

```
event = waitForEvent()
mutex.wait()
    buffer.add(event)
mutex.signal()
items.signal()
```

or,

```
mutex.wait()
    items.wait()
    event = buffer.get()
mutex.signal()
event.process()
```

Deadlock!

☠ any time you wait for a semaphore while holding a mutex!

Producer-Consumer Problem With Bounded-Buffer

Given:

```
semaphore items = 0;
semaphore spaces = BUFFER_SIZE;
```

Can we?

```
if (items >= BUFFER_SIZE)
        producer.block();
```

if: the buffer is full

then: the producer blocks until a consumer removes an item

Producer-Consumer Problem With Bounded-Buffer

Given:

```
semaphore items = 0;
semaphore spaces = BUFFER_SIZE;
```

Can we?

```
if (items >= BUFFER_SIZE)
        producer.block();
```

if: the buffer is full

then: the producer blocks until a consumer removes an item

No! We can't check the current value of a semaphore, because

! the only operations are `wait` and `signal`.

? But...

Producer-Consumer Problem With Bounded-Buffer

```
semaphore items = 0;
semaphore spaces = BUFFER_SIZE;
```

```
void producer() {
  while (true) {
    item = produceItem();
    down(spaces);
    putIntoBuffer(item);
    up(items);
  }
}
```

```
void consumer() {
  while (true) {
    down(items);
    item = rmFromBuffer();
    up(spaces);
    consumeItem(item);
  }
}
```

Producer-Consumer Problem With Bounded-Buffer

```
semaphore items = 0;
semaphore spaces = BUFFER_SIZE;
```

```
void producer() {
  while (true) {
    item = produceItem();
    down(spaces);
    putIntoBuffer(item);
    up(items);
  }
}
```

```
void consumer() {
  while (true) {
    down(items);
    item = rmFromBuffer();
    up(spaces);
    consumeItem(item);
  }
}
```

works fine when there is only one producer and one consumer, because `putIntoBuffer()` is not atomic.

`putIntoBuffer()` could contain two actions:

1. determining the next available slot
2. writing into it

Race condition:

1. Two producers decrement spaces
2. One of the producers determines the next empty slot in the buffer
3. Second producer determines the next empty slot and gets the same result as the first producer
4. Both producers write into the same slot

`putIntoBuffer()` needs to be protected with a `mutex`.

With a mutex

```
semaphore mutex = 1; //controls access to c.r.
semaphore items = 0;
semaphore spaces = BUFFER_SIZE;
```

```
void producer() {
  while (true) {
    item = produceItem();
    down(spaces);
    down(mutex);
    putIntoBuffer(item);
    up(mutex);
    up(items);
  }
}
```

```
void consumer() {
  while (true) {
    down(items);
    down(mutex);
    item = rmFromBuffer();
    up(mutex);
    up(spaces);
    consumeItem(item);
  }
}
```

9.5 Monitors

Monitors

Monitor a high-level synchronization object for achieving mutual exclusion.

- It's a language concept, and C does not have it.
- Only one process can be active in a monitor at any instant.
- It is up to the compiler to implement mutual exclusion on monitor entries.
  - The programmer just needs to know that by turning all the critical regions into monitor procedures, no two processes will ever execute their critical regions at the same time.

```
monitor example
  integer i;
  condition c;

  procedure producer();
  ...
  end;

  procedure consumer();
  ...
  end;
end monitor;
```

Monitor

The producer-consumer problem

```
monitor ProducerConsumer
  condition full, empty;
  integer count;

  procedure insert(item: integer);
  begin
    if count = N then wait(full);
    insert_item(item);
    count := count + 1;
    if count = 1 then signal(empty)
  end;

  function remove: integer;
  begin
    if count = 0 then wait(empty);
    remove = remove_item;
    count := count - 1;
    if count = N - 1 then signal(full)
  end;
  count := 0;
end monitor;
```

```
procedure producer;
begin
  while true do
  begin
    item = produce_item;
    ProducerConsumer.insert(item)
  end
end;

procedure consumer;
begin
  while true do
  begin
    item = ProducerConsumer.remove;
    consume_item(item)
  end
end;
```

9.6 Message Passing

Message Passing

- Semaphores are too low level
- Monitors are not usable except in a few programming languages
- Neither monitor nor semaphore is suitable for distributed systems
- No conflicts, easier to implement

Message passing uses two primitives, `send` and `receive` system calls:

- `send(destination, &message);`
- `receive(source,  &message);`

Message Passing

Design issues

- Message can be lost by network; — `ACK`
- What if the ACK is lost? — `SEQ`
- What if two processes have the same name? — `socket`
- Am I talking with the right guy? Or maybe a MIM? — `authentication`
- What if the sender and the receiver on the same machine? — Copying messages is always slower than doing a semaphore operation or entering a monitor.

Message Passing

TCP Header Format

<table>
<tr><th colspan="2">0</th><th colspan="2">1</th><th colspan="2">2</th><th colspan="2">3</th></tr>
<tr><td colspan="8">Source Port</td><td colspan="8">Destination Port</td></tr>
<tr><td colspan="16">Sequence Number</td></tr>
<tr><td colspan="16">Acknowledgment Number</td></tr>
<tr><td>Data Offset</td><td>0</td><td>0</td><td>0</td><td>NS</td><td>CWR</td><td>ECE</td><td>URG</td><td>ACK</td><td>PSH</td><td>RST</td><td>SYN</td><td>FIN</td><td colspan="4">Window</td></tr>
<tr><td colspan="8">Checksum</td><td colspan="8">Urgent Pointer</td></tr>
<tr><td colspan="12">Options</td><td colspan="4">Padding</td></tr>
</table>

Message Passing

The producer-consumer problem

```
#define N 100  /* number of slots in the buffer */
void producer(void)
{
  int item;
  message m;                    /* message buffer */
  while (TRUE) {
    item = produce_item();      /* generate something to put in buffer */
    receive(consumer, &m);      /* wait for an empty to arrive */
    build_message(&m, item);    /* construct a message to send */
    send(consumer, &m);         /* send item to consumer */
  }
}

void consumer(void)
{
  int item, i;
  message m;
  for (i=0; i<N; i++) send(producer, &m);  /* send N empties */
  while (TRUE) {
    receive(producer, &m);      /* get message containing item */
    item = extract_item(&m);    /* extract item from message */
    send(producer, &m);         /* send back empty reply */
    consume_item(item);         /* do something with the item */
  }
}
```

Sockets

To create a socket:

```
fd = socket(domain, type, protocol)
```

Domain Determines address format and the range of communication (local or remote). The most commonly used domains are:

Domain	Addr structure	Addr format
`AF_UNIX`	`sockaddr_un`	`/path/name`
`AF_INET`	`sockaddr_in`	`ip:port`
`AF_INET6`	`sockaddr_in6`	`ip6:port`

Type `SOCK_STREAM` (☎), `SOCK_DGRAM` (✉)

Protocol always 0

Socket System Calls

`socket()`	creates a new socket
`bind()`	binds a socket to an address (usually a well-known address on server side)
`listen()`	waits for incoming connection requests
`connect()`	sends a connection request to peer
`accept()`	accepts a connection request
`send()`/`recv()`	data transfer

Stream Sockets

Datagram Sockets

9.7 Classical IPC Problems

The Dining Philosophers Problem

```
while True:
    think()
    get_forks()
    eat()
    put_forks()
```

How to implement `get_forks()` and `put_forks()` to ensure

1. No deadlock
2. No starvation
3. Allow more than one philosopher to eat at the same time

The Dining Philosophers Problem

```
#define N 5                    /* number of philosophers */

void philosopher(int i)        /* i: philosopher number, from 0 to 4 */
{
  while (TRUE) {
    think();                   /* philosopher is thinking */
    take_fork(i);              /* take left fork */
    take_fork((i+1) % N);      /* take right fork; % is modulo operator */
    eat();                     /* yum-yum, spaghetti */
    put_fork(i);               /* put left fork back on the table */
    put_fork((i+1) % N);       /* put right fork back on the table */
  }
}
```

The Dining Philosophers Problem

```c
#define N 5                    /* number of philosophers */

void philosopher(int i)        /* i: philosopher number, from 0 to 4 */
{
  while (TRUE) {
    think();                   /* philosopher is thinking */
    take_fork(i);              /* take left fork */
    take_fork((i+1) % N);      /* take right fork; % is modulo operator */
    eat();                     /* yum-yum, spaghetti */
    put_fork(i);               /* put left fork back on the table */
    put_fork((i+1) % N);       /* put right fork back on the table */
  }
}
```

Deadlock!

? Put down the left fork and wait for a while if the right one is not available? Similar to CSMA/CD

The Dining Philosophers Problem

```c
#define N 5                    /* number of philosophers */

void philosopher(int i)        /* i: philosopher number, from 0 to 4 */
{
  while (TRUE) {
    think();                   /* philosopher is thinking */
    take_fork(i);              /* take left fork */
    take_fork((i+1) % N);      /* take right fork; % is modulo operator */
    eat();                     /* yum-yum, spaghetti */
    put_fork(i);               /* put left fork back on the table */
    put_fork((i+1) % N);       /* put right fork back on the table */
  }
}
```

Deadlock!

? Put down the left fork and wait for a while if the right one is not available? Similar to CSMA/CD — Starvation

The Dining Philosophers Problem

With One Mutex

```
#define N 5
semaphore mutex=1;

void philosopher(int i)
{
  while (TRUE) {
    think();
    wait(&mutex);
      take_fork(i);
      take_fork((i+1) % N);
      eat();
      put_fork(i);
      put_fork((i+1) % N);
    signal(&mutex);
  }
}
```

The Dining Philosophers Problem

With One Mutex

```
#define N 5
semaphore mutex=1;

void philosopher(int i)
{
  while (TRUE) {
    think();
    wait(&mutex);
      take_fork(i);
      take_fork((i+1) % N);
      eat();
      put_fork(i);
      put_fork((i+1) % N);
    signal(&mutex);
  }
}
```

☹ Only one philosopher can eat at a time.

? How about 2 mutexes? 5 mutexes?

The Dining Philosophers Problem

AST Solution (Part 1)

A philosopher may only move into eating state if neither neighbor is eating

```
#define N 5                /* number of philosophers */
#define LEFT (i+N-1)%N     /* number of i's left neighbor */
#define RIGHT (i+1)%N      /* number of i's right neighbor */
#define THINKING 0         /* philosopher is thinking */
#define HUNGRY 1           /* philosopher is trying to get forks */
#define EATING 2           /* philosopher is eating */
typedef int semaphore;
int state[N];              /* state of everyone */
semaphore mutex = 1;       /* for critical regions */
semaphore s[N];            /* one semaphore per philosopher */

void philosopher(int i)  /* i: philosopher number, from 0 to N-1 */
{
  while(TRUE) {
    think();
    take_forks(i); /* acquire two forks or block */
    eat();
    put_forks(i);  /* put both forks back on table */
  }
}
```

The Dining Philosophers Problem

AST Solution (Part 2)

```
void take_forks(int i)          /* i: philosopher number */
{
  down(&mutex);                 /* enter critical region */
  state[i] = HUNGRY;
  test(i);                      /* try to acquire 2 forks */
  up(&mutex);                   /* exit critical region */
  down(&s[i]);                  /* block if forks were not acquired */
}
void put_forks(i)               /* i: philosopher number */
{
  down(&mutex);                 /* enter critical region */
  state[i] = THINKING;
  test(LEFT);                   /* see if left neighbor can now eat */
  test(RIGHT);                  /* see if right neighbor can now eat */
  up(&mutex);                   /* exit critical region */
}
void test(i)                    /* i: philosopher number */
{
  if(state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING)
        {
      state[i] = EATING;
      up(&s[i]);
    }
}
```

The Dining Philosophers Problem

AST Solution (Part 2)

```
void take_forks(int i)          /* i: philosopher number */
{
  down(&mutex);                 /* enter critical region */
  state[i] = HUNGRY;
  test(i);                      /* try to acquire 2 forks */
  up(&mutex);                   /* exit critical region */
  down(&s[i]);                  /* block if forks were not acquired */
}
void put_forks(i)               /* i: philosopher number */
{
  down(&mutex);                 /* enter critical region */
  state[i] = THINKING;
  test(LEFT);                   /* see if left neighbor can now eat */
  test(RIGHT);                  /* see if right neighbor can now eat */
  up(&mutex);                   /* exit critical region */
}
void test(i)                    /* i: philosopher number */
{
  if(state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING)
        {
      state[i] = EATING;
      up(&s[i]);
    }
}
```

Starvation!

The Dining Philosophers Problem

More Solutions

- If there is at least one leftie and at least one rightie, then deadlock is not possible
- Wikipedia: Dining philosophers problem

The Readers-Writers Problem

Constraint: no process may access the shared data for reading or writing while another process is writing to it.

```
semaphore mutex = 1;
semaphore noOther = 1;
int readers = 0;

void writer(void)
{
  while (TRUE) {
    wait(&noOther);
      writing();
    signal(&noOther);
  }
}
```

```
void reader(void)
{
  while (TRUE) {
    wait(&mutex);
      readers++;
      if (readers == 1)
         wait(&noOther);
    signal(&mutex);
    reading();
    wait(&mutex);
      readers--;
      if (readers == 0)
        signal(&noOther);
    signal(&mutex);
    anything();
  }
}
```

The Readers-Writers Problem

Constraint: no process may access the shared data for reading or writing while another process is writing to it.

```
semaphore mutex = 1;
semaphore noOther = 1;
int readers = 0;

void writer(void)
{
  while (TRUE) {
    wait(&noOther);
      writing();
    signal(&noOther);
  }
}
```

```
void reader(void)
{
  while (TRUE) {
    wait(&mutex);
      readers++;
      if (readers == 1)
        wait(&noOther);
    signal(&mutex);
    reading();
    wait(&mutex);
      readers--;
      if (readers == 0)
        signal(&noOther);
    signal(&mutex);
    anything();
  }
}
```

Starvation!

Starvation The writer could be blocked forever if there are always someone reading.

The Readers-Writers Problem

No starvation

```
semaphore mutex = 1;
semaphore noOther = 1;
semaphore turnstile = 1;
int readers = 0;

void writer(void)
{
  while (TRUE) {
    turnstile.wait();
      wait(&noOther);
        writing();
      signal(&noOther);
    turnstile.signal();
  }
}
```

```
void reader(void)
{
  while (TRUE) {
    turnstile.wait();
    turnstile.signal();

    wait(&mutex);
      readers++;
      if (readers == 1)
         wait(&noOther);
    signal(&mutex);
    reading();
    wait(&mutex);
      readers--;
      if (readers == 0)
        signal(&noOther);
    signal(&mutex);
    anything();
  }
}
```

The Sleeping Barber Problem

Where's the problem?

- the barber saw an empty room right before a customer arrives the waiting room;
- Several customer could race for a single chair;

Solution

```c
#define CHAIRS 5
semaphore customers = 0; // any customers or not?
semaphore bber = 0;      // barber is busy
semaphore mutex = 1;
int waiting = 0;         // queueing customers
```

```c
void barber(void)
{
  while (TRUE) {
    wait(&customers);
    wait(&mutex);
       waiting--;
    signal(&mutex);
    cutHair();
    signal(&bber);
  }
}
```

```c
void customer(void)
{
  if(waiting == CHAIRS)
    goHome();
  else {
    wait(&mutex);
    waiting++;
    signal(&mutex);
    signal(&customers);
    wait(&bber);
    getHairCut();
  }
}
```

Solution2

```c
#define CHAIRS 5
semaphore customers = 0;
semaphore bber = ?;
semaphore mutex = 1;
int waiting = 0;

void barber(void)
{
  while (TRUE) {
    wait(&customers);
    cutHair();
  }
}
```

```c
void customer(void)
{
  if (waiting == CHAIRS)
    goHome();
  else {
    wait(&mutex);
    waiting++;
    signal(&mutex);
    signal(&customers);
    wait(&bber);
    getHairCut();
    signal(&bber);
    wait(&mutex);
      waiting--;
    signal(&mutex);
  }
}
```

References

Wikipedia. *Inter-process communication — Wikipedia, The Free Encyclopedia*. 2015. http://en.wikipedia.org/w/index.php?title=Inter-process%5C_communication&oldid=645037874.

Wikipedia. *Semaphore (programming) — Wikipedia, The Free Encyclopedia*. 2015. http://en.wikipedia.org/w/index.php?title=Semaphore%5C_(programming)&oldid=647556304.

10 CPU Scheduling

10.1 Process Scheduling Queues

Scheduling Queues

Job queue consists all the processes in the system

Ready queue A linked list consists processes in the main memory ready for execute

Device queue Each device has its own device queue

queue header
PCB7
PCB2
ready queue
head
tail
registers
registers
mag tape unit 0
head
tail
mag tape unit 1
head
tail
PCB3
PCB14
PCB6
disk unit 0
head
tail
PCB5
terminal unit 0
head
tail

Queueing Diagram

10.2 Scheduling

Scheduling

- Scheduler uses scheduling algorithm to choose a process from the ready queue
- Scheduling doesn't matter much on simple PCs, because
  1. Most of the time there is only one active process
  2. The CPU is too fast to be a scarce resource any more
- Scheduler has to make efficient use of the CPU because process switching is expensive
  1. User mode $\rightarrow$ kernel mode
  2. Save process state, registers, memory map...
  3. Selecting a new process to run by running the scheduling algorithm
  4. Load the memory map of the new process
  5. The process switch usually invalidates the entire memory cache

Scheduling Algorithm Goals

All systems

- **Fairness** giving each process a fair share of the CPU
- **Policy enforcement** seeing that stated policy is carried out
- **Balance** keeping all parts of the system busy

Batch systems

Throughput maximize jobs per hour

Turnaround time minimize time between submission and termination

CPU utilization keep the CPU busy all the time

Interactive systems

Response time respond to requests quickly

Proportionality meet users' expectations

Real-time systems

Meeting deadlines avoid losing data

Predictability avoid quality degradation in multimedia systems

10.3 Process Classification

Process Classification

Traditionally

CPU-bound processes vs. I/O-bound processes

Alternatively

Interactive processes responsiveness

- command shells, editors, graphical apps

Batch processes no user interaction, run in background, often penalized by the scheduler

- programming language compilers, database search engines, scientific computations

Real-time processes video and sound apps, robot controllers, programs that collect data from physical sensors

- should never be blocked by lower-priority processes
- should have a short guaranteed response time with a minimum variance

10.4 Process Behavior

Process Behavior

CPU-bound vs. I/O-bound

Types of CPU bursts:

- long bursts – CPU bound (i.e. batch work)
- short bursts – I/O bound (i.e. emacs)

As CPUs get faster, processes tend to get more I/O-bound.

10.5 Process Schedulers

Schedulers

Long-term scheduler (or job scheduler) - selects which processes should be brought into the ready queue.

Short-term scheduler (or CPU scheduler) - selects which process should be executed next and allocates CPU.

Midium-term scheduler swapping.

- LTS is responsible for a good *process mix* of I/O-bound and CPU-bound process leading to best performance.
- Time-sharing systems, e.g. UNIX, often have no long-term scheduler.

Nonpreemptive vs. preemptive

A nonpreemptive scheduling algorithm lets a process run as long as it wants until it blocks (I/O or waiting for another process) or until it voluntarily releases the CPU.

A preemptive scheduling algorithm will forcibly suspend a process after it runs for sometime. — clock interruptable

10.6 Scheduling In Batch Systems

Scheduling In Batch Systems

First-Come First-Served

- nonpreemptive
- simple
- also has a disadvantage

  What if a CPU-bound process (e.g. runs 1s at a time) followed by many I/O-bound processes (e.g. 1000 disk reads to complete)?

  - In this case, a preemptive scheduling is preferred.

Scheduling In Batch Systems

Shortest Job First

(a)

(b)

Average turnaround time

(a) $(8 + 12 + 16 + 20) \div 4 = 14$

(b) $(4 + 8 + 12 + 20) \div 4 = 11$

How to know the length of the next CPU burst?

- For long-term (job) scheduling, user provides
- For short-term scheduling, no way

10.7 Scheduling In Interactive Systems

Scheduling In Interactive Systems

Round-Robin Scheduling

- Simple, and most widely used;
- Each process is assigned a time interval, called its *quantum*;
- How long shoud the quantum be?
  - too short — too many process switches, lower CPU efficiency;
  - too long — poor response to short interactive requests;
  - usually around $20 \sim 50$ms.

Scheduling In Interactive Systems

Priority Scheduling

- SJF is a priority scheduling;
- Starvation — low priority processes may never execute;
  - Aging — as time progresses increase the priority of the process;

$ `man nice`

10.8 Thread Scheduling

Thread Scheduling

Possible: A1, A2, A3, A1, A2, A3
Not possible: A1, B1, A2, B2, A3, B3

(a)

Possible: A1, A2, A3, A1, A2, A3
Also possible: A1, B1, A2, B2, A3, B3

(b)

- With kernel-level threads, sometimes a full context switch is required
- Each process can have its own application-specific thread scheduler, which usually works better than kernel can

10.9 Linux Scheduling

Process Scheduling In Linux

A preemptive, priority-based algorithm with two separate priority ranges:

1. real-time range ($0 \sim 99$), for tasks where absolute priorities are more important than fairness
2. nice value range ($100 \sim 139$), for fair preemptive scheduling among multiple processes

In Linux, Process Priority is Dynamic

The scheduler keeps track of what processes are doing and adjusts their priorities periodically

- Processes that have been denied the use of a CPU for a long time interval are boosted by dynamically increasing their priority (usually I/O-bound)
- Processes running for a long time are penalized by decreasing their priority (usually CPU-bound)
- Priority adjustments are performed only on user tasks, not on real-time tasks

Tasks are determined to be I/O-bound or CPU-bound based on an interactivity heuristic

A task's interactiveness metric is calculated based on how much time the task executes compared to how much time it sleeps

Problems With The Pre-2.6 Scheduler

- 😐 an algorithm with $O(n)$ complexity
- a single runqueue for all processors
  - ☺ good for load balancing
  - ☹ bad for CPU caches, when a task is rescheduled from one CPU to another
- 😐 a single runqueue lock — only one CPU working at a time

Scheduling In Linux 2.6 Kernel

- ☺ $O(1)$ — Time for finding a task to execute depends not on *the number of active tasks* but instead on *the number of priorities*
- ☺ Each CPU has its own *runqueue*, and schedules itself independently; better cache efficiency
- The job of the scheduler is simple — Choose the task on the highest priority list to execute

How to know there are processes waiting in a priority list?

A priority bitmap (5 32-bit words for 140 priorities) is used to define when tasks are on a given priority list.

- `find-first-bit-set` instruction is used to find the highest priority bit.

Each runqueue has two priority arrays

Completely Fair Scheduling (CFS)

Linux's Process Scheduler

up to 2.4: simple, scaled poorly

- $O(n)$
- non-preemptive in kernel mode
- single run queue (cache? SMP?)

from 2.5 on: $O(1)$ scheduler

- ☺ 140 priority lists — scaled well
- ☺ one run queue per CPU — true SMP support
- ☺ preemptive
- ☺ ideal for large server workloads
- ☹ showed latency on desktop systems

from 2.6.23 on: Completely Fair Scheduler (CFS)

- ☺ improved interactive performance

Completely Fair Scheduler (CFS)

For a perfect (unreal) multitasking CPU

- n runnable processes can run at the same time
- each process should receive $\frac{1}{n}$ of CPU power

For a real world CPU

- can run only a single task at once — unfair
  - ☺ while one task is running
  - ☹ the others have to wait
- `p->wait_runtime` is the amount of time the task should now run on the CPU for it becomes completely fair and balanced.
  - ☺ on ideal CPU, the `p->wait_runtime` value would always be zero
- CFS always tries to run the task with the largest `p->wait_runtime` value

CFS

In practice it works like this:

- While a task is using the CPU, its `wait_runtime` decreases

```
wait_runtime = wait_runtime - time_running
```

if: its `wait_runtime` $\neq$ $\mathtt{MAX}_{\mathtt{wait_runtime}}$ (among all processes)

then: it gets preempted

- Newly woken tasks (`wait_runtime = 0`) are put into the tree more and more to the right
- slowly but surely giving a chance for every task to become the "leftmost task" and thus get on the CPU within a deterministic amount of time

References

Wikipedia. *Scheduling (computing) — Wikipedia, The Free Encyclopedia*. 2015. http://en.wikipedia.org/w/index.php?title=Scheduling%5C_(computing)&oldid=647892123.

11 Deadlock

11.1 Resources

A Major Class of Deadlocks Involve Resources

Processes need access to resources in reasonable order

Suppose...

- a process holds resource A and requests resource B. At same time,
- another process holds B and requests A

Both are blocked and remain so

Examples of computer resources

- printers
- memory space
- data (e.g. a locked record in a DB)
- semaphores

Resources

```
typedef int semaphore;
    semaphore resource_1;
    semaphore resource_2;

    void process_A(void) {
        down(&resource_1);
        down(&resource_2);
        use_both_resources( );
        up(&resource_2);
        up(&resource_1);
    }

    void process_B(void) {
        down(&resource_1);
        down(&resource_2);
        use_both_resources( );
        up(&resource_2);
        up(&resource_1);
    }
```

```
semaphore resource_1;
semaphore resource_2;

void process_A(void) {
    down(&resource_1);
    down(&resource_2);
    use_both_resources( );
    up(&resource_2);
    up(&resource_1);
}

void process_B(void) {
    down(&resource_2);
    down(&resource_1);
    use_both_resources( );
    up(&resource_1);
    up(&resource_2);
}
```

Deadlock!

Resources

Deadlocks occur when ...

processes are granted exclusive access to *resources*

e.g. devices, data records, files, ...

Preemptable resources can be taken away from a process with no ill effects

e.g. memory

Nonpreemptable resources will cause the process to fail if taken away

e.g. CD recorder

In general, deadlocks involve nonpreemptable resources.

Resources

Sequence of events required to use a resource

request	➠	use	➠	release
`open()`				`close()`
`allocate()`				`free()`

What if request is denied?

Requesting process

- may be blocked
- may fail with error code

11.2 Introduction to Deadlocks

The Best Illustration of a Deadlock

A law passed by the Kansas legislature early in the 20th century

"When two trains approach each other at a crossing, both shall come to a full stop and neither shall start up again until the other has gone."

Introduction to Deadlocks

Deadlock

A set of processes is deadlocked if each process in the set is waiting for an event that only another process in the set can cause.

- Usually the event is release of a currently held resource
- None of the processes can ...
  - run
  - release resources
  - be awakened

Four Conditions For Deadlocks

Mutual exclusion condition each resource can only be assigned to one process or is available

Hold and wait condition process holding resources can request additional

No preemption condition previously granted resources cannot forcibly taken away

Circular wait condition

- must be a circular chain of 2 or more processes
- each is waiting for resource held by next member of the chain

Four Conditions For Deadlocks

Unlocking a deadlock is to answer 4 questions:

1. Can a resource be assigned to more than one process at once?
2. Can a process hold a resource and ask for another?
3. can resources be preempted?
4. Can circular waits exits?

11.3 Deadlock Modeling

Resource-Allocation Graph

Strategies for dealing with Deadlocks

1. detection and recovery
2. dynamic avoidance — careful resource allocation
3. prevention — negating one of the four necessary conditions
4. just ignore the problem altogether

Resource-Allocation Graph

Resource-Allocation Graph

Basic facts:

- No cycles ➠ no deadlock
- If graph contains a cycle ➠
  - if only one instance per resource type, then deadlock
  - if several instances per resource type, possibility of deadlock

11.4 Deadlock Detection and Recovery

Deadlock Detection

- Allow system to enter deadlock state
- Detection algorithm
- Recovery scheme

Deadlock Detection

Single Instance of Each Resource Type — Wait-for Graph

- $\boxed{P_i \rightarrow P_j}$ if P_i is waiting for P_j;
- Periodically invoke an algorithm that searches for a cycle in the graph. If there is a cycle, there exists a deadlock.
- An algorithm to detect a cycle in a graph requires an order of n^2 operations, where n is the number of vertices in the graph.

Deadlock Detection

Several Instances of a Resource Type

$$E = \begin{array}{cccc} \text{Tape drives} & \text{Plotters} & \text{Scanners} & \text{CD Roms} \\ (\,4 & 2 & 3 & 1\,) \end{array}$$

$$A = \begin{array}{cccc} \text{Tape drives} & \text{Plotters} & \text{Scanners} & \text{CD Roms} \\ (\,2 & 1 & 0 & 0\,) \end{array}$$

Current allocation matrix

$$C = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 2 & 0 & 0 & 1 \\ 0 & 1 & 2 & 0 \end{bmatrix}$$

Request matrix

$$R = \begin{bmatrix} 2 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 2 & 1 & 0 & 0 \end{bmatrix}$$

Row n:

- C: current allocation to process n
- R: current requirement of process n

Column m:

- C: current allocation of resource class m
- R: current requirement of resource class m

Deadlock Detection

Several Instances of a Resource Type

Resources in existence
$(E_1, E_2, E_3, \ldots, E_m)$

Resources available
$(A_1, A_2, A_3, \ldots, A_m)$

Current allocation matrix

$$\begin{bmatrix} C_{11} & C_{12} & C_{13} & \cdots & C_{1m} \\ C_{21} & C_{22} & C_{23} & \cdots & C_{2m} \\ \vdots & \vdots & \vdots & & \vdots \\ C_{n1} & C_{n2} & C_{n3} & \cdots & C_{nm} \end{bmatrix}$$

Row n is current allocation to process n

Request matrix

$$\begin{bmatrix} R_{11} & R_{12} & R_{13} & \cdots & R_{1m} \\ R_{21} & R_{22} & R_{23} & \cdots & R_{2m} \\ \vdots & \vdots & \vdots & & \vdots \\ R_{n1} & R_{n2} & R_{n3} & \cdots & R_{nm} \end{bmatrix}$$

Row 2 is what process 2 needs

$$\sum_{i=1}^{n} C_{ij} + A_j = E_j$$

e.g.

$$(C_{13} + C_{23} + \ldots + C_{n3}) + A_3 = E_3$$

Maths recall: vectors comparison

For two vectors, X and Y

$$X \leq Y \quad \text{iff} \quad X_i \leq Y_i \quad \text{for} \quad 0 \leq i \leq m$$

e.g.

$$\begin{bmatrix} 1 & 2 & 3 & 4 \end{bmatrix} \leq \begin{bmatrix} 2 & 3 & 4 & 4 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 2 & \mathbf{3} & 4 \end{bmatrix} \nleq \begin{bmatrix} 2 & 3 & \mathbf{2} & 4 \end{bmatrix}$$

Deadlock Detection

Several Instances of a Resource Type

	Tape drives	Plotters	Scanners	CD Roms
E = (	4	2	3	1)

	Tape drives	Plotters	Scanners	CD Roms
A = (	2	1	0	0)

Current allocation matrix

$$C = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 2 & 0 & 0 & 1 \\ 0 & 1 & 2 & 0 \end{bmatrix}$$

Request matrix

$$R = \begin{bmatrix} 2 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 2 & 1 & 0 & 0 \end{bmatrix}$$

Deadlock Detection

Several Instances of a Resource Type

	Tape drives	Plotters	Scanners	CD Roms
E = (	4	2	3	1)

	Tape drives	Plotters	Scanners	CD Roms
A = (	2	1	0	0)

$$\overset{A}{(2\ 1\ 0\ 0)} \geq \overset{R}{R_3}, (2\ 1\ 0\ 0)$$

Current allocation matrix

$$C = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 2 & 0 & 0 & 1 \\ 0 & 1 & 2 & 0 \end{bmatrix}$$

Request matrix

$$R = \begin{bmatrix} 2 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 2 & 1 & 0 & 0 \end{bmatrix}$$

Deadlock Detection

Several Instances of a Resource Type

	Tape drives	Plotters	Scanners	CD Roms
E = (	4	2	3	1)

	Tape drives	Plotters	Scanners	CD Roms
A = (	2	1	0	0)

$$\overset{A}{(2\ 1\ 0\ 0)} \geq \overset{R}{R_3}, (2\ 1\ 0\ 0)$$
$$(2\ 2\ 2\ 0) \geq R_2, (1\ 0\ 1\ 0)$$

Current allocation matrix

$$C = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 2 & 0 & 0 & 1 \\ 0 & 1 & 2 & 0 \end{bmatrix}$$

Request matrix

$$R = \begin{bmatrix} 2 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 2 & 1 & 0 & 0 \end{bmatrix}$$

Deadlock Detection

Several Instances of a Resource Type

	Tape drives	Plotters	Scanners	CD Roms
E = (	4	2	3	1)

	Tape drives	Plotters	Scanners	CD Roms
A = (	2	1	0	0)

$$\begin{array}{ccc} A & & R \\ (2\ 1\ 0\ 0) & \geq & R_3, (2\ 1\ 0\ 0) \\ (2\ 2\ 2\ 0) & \geq & R_2, (1\ 0\ 1\ 0) \\ (4\ 2\ 2\ 1) & \geq & R_1, (2\ 0\ 0\ 1) \end{array}$$

Current allocation matrix

$$C = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 2 & 0 & 0 & 1 \\ 0 & 1 & 2 & 0 \end{bmatrix}$$

Request matrix

$$R = \begin{bmatrix} 2 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 2 & 1 & 0 & 0 \end{bmatrix}$$

Recovery From Deadlock

- Recovery through preemption
  - take a resource from some other process
  - depends on nature of the resource
- Recovery through rollback
  - checkpoint a process periodically
  - use this saved state
  - restart the process if it is found deadlocked
- Recovery through killing processes

11.5 Deadlock Avoidance

Deadlock Avoidance

Resource Trajectories

- *B* is requesting a resource at point `t`. The system must decide whether to grant it or not.
- Deadlock is unavoidable if you get into unsafe region.

Deadlock Avoidance

Safe and Unsafe States

Assuming $E = 10$

Unsafe

	Has	Max
A	3	9
B	2	4
C	2	7

Free: 3

(a)

	Has	Max
A	4	9
B	2	4
C	2	7

UNSAFE

Free: 2

(b)

	Has	Max
A	4	9
B	4	4
C	2	7

Free: 0

(c)

	Has	Max
A	4	9
B	—	—
C	2	7

Free: 4

(d)

Safe

	Has	Max
A	3	9
B	2	4
C	2	7

Free: 3

(a)

	Has	Max
A	3	9
B	4	4
C	2	7

Free: 1

(b)

	Has	Max
A	3	9
B	0	–
C	2	7

Free: 5

(c)

	Has	Max
A	3	9
B	0	–
C	7	7

Free: 0

(d)

	Has	Max
A	3	9
B	0	–
C	0	–

Free: 7

(e)

Deadlock Avoidance

The Banker's Algorithm for a Single Resource

The banker's algorithm considers each request as it occurs, and sees if granting it leads to a safe state.

	Has	Max
A	0	6
B	0	5
C	0	4
D	0	7

Free: 10

(a)

	Has	Max
A	1	6
B	1	5
C	2	4
D	4	7

Free: 2

(b)

	Has	Max
A	1	6
B	2	5
C	2	4
D	4	7

UNSAFE!

Free: 1

(c)

$$unsafe \neq deadlock$$

Deadlock Avoidance

The Banker's Algorithm for Multiple Resources

Process	Tape drives	Plotters	Scanners	CD ROMs
A	3	0	1	1
B	0	1	0	0
C	1	1	1	0
D	1	1	0	1
E	0	0	0	0

Resources assigned

Process	Tape drives	Plotters	Scanners	CD ROMs
A	1	1	0	0
B	0	1	1	2
C	3	1	0	0
D	0	0	1	0
E	2	1	1	0

Resources still needed

E = (6342)
P = (5322)
A = (1020)

$$D \rightarrow A \rightarrow B, C, E$$

$$D \rightarrow E \rightarrow A \rightarrow B, C$$

Deadlock Avoidance

Mission Impossible

In practice

- processes rarely know in advance their max future resource needs;
- the number of processes is not fixed;
- the number of available resources is not fixed.

Conclusion: Deadlock avoidance is essentially a mission impossible.

11.6 Deadlock Prevention

Deadlock Prevention

Break The Four Conditions

Attacking the Mutual Exclusion Condition

- For example, using a *printing daemon* to avoid exclusive access to a printer.
- Not always possible
  - Not required for sharable resources;
  - must hold for nonsharable resources.
- The best we can do is to avoid mutual exclusion as much as possible
  - Avoid assigning a resource when not really necessary
  - Try to make sure as few processes as possible may actually claim the resource

Attacking the Hold and Wait Condition

Must guarantee that whenever a process requests a resource, it does not hold any other resources.

Try: the processes must request all their resources before starting execution

if everything is available
then can run
if one or more resources are busy
then nothing will be allocated (just wait)

Problem:

- many processes don't know what they will need before running
- Low resource utilization; starvation possible

Attacking the No Preemption Condition

if a process that is holding some resources requests another resource that cannot be immediately allocated to it

then
1. All resources currently being held are released
2. Preempted resources are added to the list of resources for which the process is waiting
3. Process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting

Low resource utilization; starvation possible

Attacking Circular Wait Condition

Impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration

1. Imagesetter
2. Scanner
3. Plotter
4. Tape drive
5. CD Rom drive

(a)

(b)

It's hard to find an ordering that satisfies everyone.

11.7 The Ostrich Algorithm

The Ostrich Algorithm

- Pretend there is no problem
- Reasonable if
  - deadlocks occur very rarely
  - cost of prevention is high
- UNIX and Windows takes this approach
- It is a trade off between
  - convenience
  - correctness

References

Wikipedia. *Deadlock — Wikipedia, The Free Encyclopedia*. 2015. http://en.wikipedia.org/w/index.php?title=Deadlock&oldid=645602687.

Part III

Memory Management

12 Background

Memory Management

In a perfect world Memory is *large, fast, non-volatile*

In real world ...

Memory manager handles the memory hierarchy.

Basic Memory Management

Real Mode

In the old days ...

- Every program simply saw the physical memory
- mono-programming without swapping or paging

Basic Memory Management

Relocation Problem

EXPOSING PHYSICAL MEMORY TO PROCESSES IS NOT A GOOD IDEA

Memory Protection

Protected mode

We need

- Protect the OS from access by user programs
- Protect user programs from one another

Protected mode is an operational mode of x86-compatible CPU.

- The purpose is to protect everyone else (including the OS) from your program.

Memory Protection

Logical Address Space

Base register holds the smallest legal physical memory address

Limit register contains the size of the range

A pair of `base` and `limit` registers define the logical address space

Memory Protection

Base and limit registers

UNIX View of a Process' Memory

text: program code

data: initialized global and static data

bss: uninitialized global and static data

heap: dynamically allocated with `malloc, new`

stack: local variables

Stack vs. Heap

Stack	Heap
compile-time allocation	run-time allocation
auto clean-up	you clean-up
inflexible	flexible
smaller	bigger
quicker	slower

How large is the ...

stack: `ulimit -s`

heap: could be as large as your virtual memory

text|data|bss: `size a.out`

Multi-step Processing of a User Program

When is space allocated?

Static: before program start running

- Compile time
- Load time

Dynamic: as program runs

- Execution time

Address Binding

Who assigns memory to segments?

Static-binding: before a program starts running

Compile time: Compiler and assembler generate an object file for each source file

Load time:

- Linker combines all the object files into a single executable object file
- Loader (part of OS) loads an executable object file into memory at location(s) determined by the OS
  - invoked via the `execve` system call

Dynamic-binding: as program runs

- Execution time:
  - uses `new` and `malloc` to dynamically allocate memory
  - gets space on stack during function calls

Static loading

- The entire program and all data of a process must be in physical memory for the process to execute
- The size of a process is thus limited to the size of physical memory

Dynamic Linking

A dynamic linker is actually a special loader that loads external shared libraries into a running process

- Small piece of code, `stub`, used to locate the appropriate memory-resident library routine
- Only one copy in memory
- Don't have to re-link after a library update

```
if ( stub_is_executed ){
  if ( !routine_in_memory )
     load_routine_into_memory();
  stub_replaces_itself_with_routine();
  execute_routine();
}
```

Dynamic Linking

Library Files

Static libraries `.a` files. Very old ones, but still alive.

```
$ find /usr/lib -name "*.a"
```

Shared libraries `.so` files. The preferred ones.

```
$ find /usr/lib -name "*.so.*"
```

Examples:

```
$ gcc -o hello hello.c /usr/lib/libm.a
$ gcc -o hello hello.c -lm
```

Build A Static Library

Source codes

main.c

```c
#include "lib.h"

int main(int argc, char* argv[])
{
  int i=1;

  for (; i<argc; i++)
    {
      hello(argv[i]);
      hi(argv[i]);
    }
  return 0;
}
```

lib.h

```c
#include <stdio.h>

void hello(char *);
void hi(char *);
```

hello.c

```c
#include <stdio.h>

void hello(char *arg)
{
  printf("Hello, %s!\n", arg);
}
```

hi.c

```c
#include <stdio.h>

void hi(char *arg)
{
  printf("Hi, %s!\n", arg);
}
```

Build A Static Library

Step by step

1. Get `hello.o` and `hi.o`

   ```
   $ gcc -c hello.c hi.c
   ```

2. Put `*.o` into `libhi.a`

   ```
   $ ar crv libhi.a hello.o hi.o
   ```

3. Use `libhi.a`

   ```
   $ gcc main.c libhi.a
   ```

Build A Static Library

Makefile

```
main: main.c lib.h libhi.a
        gcc -Wall -o main main.c libhi.a

libhi.a: hello.o hi.o
        ar crv libhi.a hello.o hi.o

hello.o: hello.c
        gcc -Wall -c hello.c

hi.o: hi.c
        gcc -Wall -c hi.c

clean:
        rm -f *.o *.a main
```

Build A Shared Library

Source codes

hello.c

```c
#include "hello.h"

int main(int argc, char *argv[])
{
  if (argc != 2)
    printf ("Usage: %s needs an argument.\n", argv[0]);
  else
    hi(argv[1]);
  return 0;
}
```

hello.h

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

int hi(char*);
```

hi.c

```c
#include "hello.h"

int hi(char* s)
{
  printf ("Hi, %s\n",s);
  return 0;
}
```

Build A Shared Library

Step by step

1. Get `hi.o`

   ```
   $ gcc -fPIC -c hi.c
   ```

2. Get `libhi.so`

   ```
   $ gcc -shared -o libhi.so hi.o
   ```

3. Use `libhi.so`

   ```
   $ gcc -L. -Wl,-rpath=. hello.c -lhi
   ```

4. Check it

   ```
   $ ldd a.out
   ```

Build A Shared Library

Makefile

```
# http://www.cprogramming.com/tutorial/shared-libraries-linux-gcc.html
# http://tldp.org/HOWTO/Program-Library-HOWTO/shared-libraries.html
#
# gcc -fPIC -c hi.c
# gcc -shared -o libhi.so hi.o
# gcc -L/current/dir -Wl,option -Wall -o hello hello.c -lhi
#
# -L          - tells ld where to search libraries
# -Wl,option - pass option as an option to the linker (ld)
# -rpath=dir - Add a directory to the runtime library search path

hello: hello.c hello.h libhi.so
        gcc -L. -Wl,-rpath=. -Wall -o hello hello.c -lhi
libhi.so: hi.o hello.h
        gcc -shared -o libhi.so hi.o
hi.o: hi.c hello.h
        gcc -fPIC -c hi.c
clean:
        rm *.o *.so hello
```

GNU C Library

Linux API > POSIX API

```
$ man 7 libc
$ man 3 intro
$ man gcc
$ info gcc
sudo apt install gcc-doc
```

Logical vs. Physical Address Space

- Mapping logical address space to physical address space is central to MM

  Logical address generated by the CPU; also referred to as virtual address

  Physical address address seen by the memory unit
- In compile-time and load-time address binding schemes, LAS and PAS are identical in size
- In execution-time address binding scheme, they are differ.

Logical vs. Physical Address Space

The user program

- deals with logical addresses
- never sees the real physical addresses

MMU

Memory Management Unit

Memory Protection

Swapping

Major part of swap time is transfer time

Total transfer time is directly proportional to the amount of memory swapped

13 Contiguous Memory Allocation

Contiguous Memory Allocation

Multiple-partition allocation

Operating system maintains information about:

a allocated partitions

b free partitions (hole)

Dynamic Storage-Allocation Problem

First Fit, Best Fit, Worst Fit

First-fit: The first hole that is big enough

Best-fit: The smallest hole that is big enough

- Must search entire list, unless ordered by size
- Produces the smallest leftover hole

Worst-fit: The largest hole

- Must also search entire list
- Produces the largest leftover hole

- First-fit and best-fit better than worst-fit in terms of speed and storage utilization
- First-fit is generally faster

Fragmentation

Reduce external fragmentation by

- Compaction is possible only if relocation is dynamic, and is done at execution time
- Noncontiguous memory allocation
  - Paging
  - Segmentation

14 Virtual Memory

Virtual Memory

Logical memory can be much larger than physical memory

Address translation

$$\textit{virtual address} \xrightarrow{\textit{page table}} \textit{physical address}$$

$$\textit{Page } 0 \xrightarrow{\textit{map to}} \textit{Frame } 2$$

$$0_{\textit{virtual}} \xrightarrow{\textit{map to}} 8192_{\textit{physical}}$$

$$\begin{array}{c} 20500_{\textit{vir}} \\ (20k+20)_{\textit{vir}} \end{array} \xrightarrow{\textit{map to}} \begin{array}{c} 12308_{\textit{phy}} \\ (12k+20)_{\textit{phy}} \end{array}$$

14.1 Paging

Paging

Address Translation Scheme

Address generated by CPU is divided into:

Page number(p): an index into a *page table*

Page offset(d): to be copied into memory

Given logical address space (2^m) and page size (2^n),

$$\text{number of pages} = \frac{2^m}{2^n} = 2^{m-n}$$

Example: addressing to 0010000000000100

$$\underbrace{\overbrace{0\,0\,1\,0}^{m-n=4}\ \overbrace{0\,0\,0\,0\,0\,0\,0\,0\,0\,1\,0\,0}^{n=12}}_{m=16}$$

$$\text{page number} = 0010 = 2, \quad \text{page offset} = 000000000100$$

Virtual pages:	16
Page size:	4k
Virtual memory:	64K
Physical frames:	8
Physical memory:	32K

Page Fault

`MOV REG, 32780`?

➠ Page fault & swapping

Page Fault Handling

Shared Pages

Page Table Entry

Intel i386 Page Table Entry

- Commonly 4 bytes (32 bits) long
- Page size is usually 4k (2^{12} bytes). OS dependent

  `$ getconf PAGESIZE`
- Could have $2^{32-12} = 2^{20} = 1M$ pages

  Could address $1M \times 4KB = 4GB$ memory

```
  P – PRESENT
R/W – READ/WRITE
U/S – USER/SUPERVISOR
  A – ACCESSED
  D – DIRTY
AVAIL – AVAILABLE FOR SYSTEMS PROGRAMMER USE

NOTE: 0 INDICATES INTEL RESERVED. DO NOT DEFINE.
```

Page Table

- Page table is kept in main memory
- Usually one page table for each process
- Page-table base register (PTBR): A pointer to the page table is stored in PCB
- Page-table length register (PRLR): indicates size of the page table
- Slow
  - Requires two memory accesses. One for the page table and one for the data/instruction.
- TLB

Translation Lookaside Buffer (TLB)

80-20 rule Only a small fraction of the PTEs are heavily read; the rest are barely used at all

Multilevel Page Tables

- a $1M$-entry page table eats $4M$ memory
- while 100 processes running, $400M$ memory is gone for page tables
- avoid keeping all the page tables in memory all the time

A two-level scheme

```
        page number          page offset
 ┌──────────────────────┬──────────────────┐
 │        p1            │       p2         │        d         │
 └──────────────────────┴──────────────────┘

     10                10                 12
      │                 │
      │                 └──> pointing to 1k frames
      └────> pointing to 1k page tables
```

Two-Level Page Tables

Example

Problem With 64-bit Systems

Given:

- virtual address space $= 64\,bits$
- page size $= 4\,KB = 2^{12}\,B$

? How much space would a simple single-level page table take?

if Each page table entry takes $4\,Bytes$

then The whole page table (2^{64-12} entries) will take

$$2^{64-12} \times 4\,B = 2^{54}\,B = 16\,PB \quad (peta \Rightarrow tera \Rightarrow giga)!$$

And this is for ONE process!

Multi-level?

if $10\,bits$ for each level

then $\frac{64-12}{10} = 5$ levels are required

5 memory accress for each address translation!

Inverted Page Tables

Index with frame number

Inverted Page Table:

- One entry for each *physical* frame
  - The physical frame number is the table index
- A single global page table for all processes
  - The table is shared — PID is required
- Physical pages are now mapped to virtual — each entry contains a virtual page number instead of a physical one
- Information bits, e.g. protection bit, are as usual

Find index according to entry contents

$(pid, p) \Rightarrow i$

Std. PTE (32-bit sys.):

page frame address	info
20	12

indexed by page number

if 2^{20} *entries*, $4\,B$ each

then $SIZE_{page\ table} = 2^{20} \times 4 = 4\,MB$ (for *each* process)

Inverted PTE (64-bit sys.):

pid	virtual page number	info
16	52	12

indexed by frame number

if assuming

- 16 bits for PID
- 52 bits for virtual page number
- 12 bits of information

then each entry takes $16 + 52 + 12 = 80\,bits = 10\,bytes$

if physical mem $= 1G\,(2^{30}\,B)$, and page size $= 4K\,(2^{12}\,B)$, we'll have $2^{30-12} = 2^{18}$ *pages*

then $SIZE_{page\ table} = 2^{18} \times 10\,B = 2.5\,MB$ (for *all* processes)

Inefficient: Require searching the entire table

Hashed Inverted Page Tables

A hash anchor table — an extra level before the actual page table

- maps $\begin{matrix}\text{process IDs}\\ \text{virtual page numbers}\end{matrix}$ $\Rightarrow$ page table entries
- Since collisions may occur, the page table must do chaining

Hashed Inverted Page Table

14.2 Demand Paging

Demand Paging

With demand paging, the size of the LAS is no longer constrained by physical memory

- Bring a page into memory only when it is needed
  - Less I/O needed
  - Less memory needed
  - Faster response
  - More users
- Page is needed $\Rightarrow$ reference to it
  - invalid reference $\Rightarrow$ abort
  - not-in-memory $\Rightarrow$ bring to memory
- Lazy swapper - never swaps a page into memory unless page will be needed
  - Swapper deals with entire processes
  - Pager (Lazy swapper) deals with pages

14.3 Copy-on-Write

Copy-on-Write

More efficient process creation

- Parent and child processes initially share the same pages in memory
- Only the modified page is copied upon modification occurs
- Free pages are allocated from a *pool* of zeroed-out pages

14.4 Memory mapped files

Memory Mapped Files

Mapping a file (disk block) to one or more memory pages

- Improved I/O performance — much faster than *read()* and *write()* system calls
- Lazy loading (demand paging) — only a small portion of file is loaded initially
- A mapped file can be shared, like *shared library*

14.5 Page Replacement Algorithms

Need For Page Replacement

Page replacement: find some page in memory, but not really in use, swap it out

Performance Concern

Because disk I/O is so expensive, we must solve two major problems to implement demand paging.

Frame-allocation algorithm If we have multiple processes in memory, we must decide how many frames to allocate to each process.

Page-replacement algorithm When page replacement is required, we must select the frames that are to be replaced.

Performance

We want an algorithm resulting in lowest page-fault rate

- Is the victim page modified?
- Pick a random page to swap out?
- Pick a page from the faulting process' own pages? Or from others?

Page-Fault Frequency Scheme

Establish ”acceptable” page-fault rate

FIFO Page Replacement Algorithm

reference string

7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
7	7	7	2		2	2	4	4	4	0			0	0			7	7	7
	0	0	0		3	3	3	2	2	2			1	1			1	0	0
		1	1		1	0	0	0	3	3			3	2			2	2	1

page frames

- Maintain a linked list (FIFO queue) of all pages
  - in order they came into memory
- Page at beginning of list replaced
- Disadvantage
  - The oldest page may be often used
  - Belady's anomaly

FIFO Page Replacement Algorithm

Belady's Anomaly

- Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
- 3 frames (3 pages can be in memory at a time per process)

$\boxed{1}$	4	5	3
$\boxed{2}$	1	~~1~~	4
$\boxed{3}$	2	~~2~~	~~5~~

9 page faults

- 4 frames

$\boxed{1}$	~~1~~	2
$\boxed{2}$	~~2~~	3
$\boxed{3}$	5	4
$\boxed{4}$	1	5

10 page faults

- Belady's Anomaly: more frames $\Rightarrow$ more page faults

Optimal Page Replacement Algorithm (OPT)

reference string

7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
7	7	7	2		2		2			2			2				7		
	0	0	0		0		4			0			0				0		
		1	1		3		3			3			1				1		

page frames

- Replace page needed at the farthest point in future
  - Optimal but not feasible
- Estimate by ...
  - logging page use on previous runs of process
  - although this is impractical, similar to SJF CPU-scheduling, it can be used for comparison studies

Least Recently Used (LRU) Algorithm

FIFO uses the time when a page was brought into memory

OPT uses the time when a page is to be used

LRU uses the recent past as an approximation of the near future

Replace the page that *has not been used* for the longest period of time

Assume recently-used-pages will used again soon

reference string

7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
7	7	7	2		2		4	4	4	0			1		1		1		
	0	0	0		0		0	0	3	3			3		0		0		
		1	1		3		3	2	2	2			2		2		7		

page frames

LRU Implementations

Counters: record the time of the last reference to each page

- choose page with lowest value counter
- Keep counter in each page table entry
- counter overflow — periodically zero the counter
- require a search of the page table to find the LRU page
- update time-of-use field in the page table every memory reference!

Stack: keep a linked list (stack) of pages

- most recently used at top, least (LRU) at bottom
  - no search for replacement
- whenever a page is referenced, it's removed from the stack and put on the top
  - update this list every memory reference!

Second Chance Page Replacement Algorithm

When a page fault occurs, the page the hand is pointing to is inspected. The action taken depends on the R bit:

- R = 0: Evict the page
- R = 1: Clear R and advance hand

14.6 Allocation of Frames

Allocation of Frames

- Each process needs minimum number of pages
- Fixed Allocation
  - Equal allocation — e.g., 100 frames and 5 processes, give each process 20 frames.
  - Proportional allocation — Allocate according to the size of process

$$a_i = \frac{s_i}{\sum s_i} \times m$$

S_i: size of process p_i
m: total number of frames
a_i: frames allocated to p_i

- Priority Allocation — Use a proportional allocation scheme using priorities rather than size

$$\frac{priority_i}{\sum priority_i} \quad or \quad (\frac{s_i}{\sum s_i}, \frac{priority_i}{\sum priority_i})$$

Global vs. Local Allocation

If process P_i generates a page fault, it can select a replacement frame

- from its own frames — Local replacement
- from the set of all frames; one process can take a frame from another — Global replacement
  - from a process with lower priority number

Global replacement generally results in greater system throughput.

14.7 Thrashing And Working Set Model

Thrashing

Thrashing

1. CPU not busy $\Rightarrow$ add more processes
2. a process needs more frames $\Rightarrow$ faulting, and taking frames away from others
3. these processes also need these pages $\Rightarrow$ also faulting, and taking frames away from others $\Rightarrow$ chain reaction
4. more and more processes queueing for the paging device $\Rightarrow$ ready queue is empty $\Rightarrow$ CPU has nothing to do $\Rightarrow$ add more processes $\Rightarrow$ more page faults
5. MMU is busy, but no work is getting done, because processes are busy paging — thrashing

Demand Paging and Thrashing

Locality Model

- A locality is a set of pages that are actively used together
- Process migrates from one locality to another

locality in a memory reference pattern

Why does thrashing occur?

$$\sum_{i=(0,n)} Locality_i > total\ memory\ size$$

Working-Set Model

Working Set (WS) The set of pages that a process is currently(Δ) using. ($\approx$ locality)

Δ: Working-set window. In this example,

$$\Delta = 10 \texttt{ memory access}$$

WSS: Working-set size. $WS(t_1) = \{\overbrace{1,\ 2,\ 5,\ 6,\ 7}^{WSS=5}\}$

- The accuracy of the working set depends on the selection of Δ
- Thrashing, if $\sum WSS_i > SIZE_{total\,memory}$

The Working-Set Page Replacement Algorithm

To evict a page that is not in the working set

$$age = Current\ virtual\ time - Time\ of\ last\ use$$

The WSClock Page Replacement Algorithm

Combine Working Set Algorithm With Clock Algorithm

14.8 Other Issues

Other Issues — Prepaging

- reduce faulting rate at (re)startup
  - remember working-set in PCB
- Not always work
  - if prepaged pages are unused, I/O and memory was wasted

Other Issues — Page Size

Larger page size

- ☹ Bigger internal fragmentation
- ☹ longer I/O time

Smaller page size

- ☹ Larger page table
- ☹ more page faults
  - one page fault for each byte, if page size $= 1$ Byte
  - for a $200K$ process, with page size $= 200K$, only one page fault

No best answer

$ `getconf PAGESIZE`

Other Issues — TLB Reach

- Ideally, the working set of each process is stored in the TLB
  - Otherwise there is a high degree of page faults
- TLB Reach — The amount of memory accessible from the TLB

$$\textit{TLB Reach} = (\textit{TLB Size}) \times (\textit{Page Size})$$

- Increase the page size

  Internal fragmentation may be increased
- Provide multiple page sizes
  - This allows applications that require larger page sizes the opportunity to use them without an increase in fragmentation
    - UltraSPARC supports page sizes of $8KB$, $64KB$, $512KB$, and $4MB$
    - Pentium supports page sizes of $4KB$ and $4MB$

Other Issues — Program Structure

Careful selection of data structures and programming structures can increase locality, i.e. lower the page-fault rate and the number of pages in the working set.

Example

- A stack has a good locality, since access is always made to the top
- A hash table has a bad locality, since it's designed to scatter references
- Programming language
  - Pointers tend to randomize access to memory
  - OO programs tend to have a poor locality

Other Issues — Program Structure

Example

- `int[i][j] = int[128][128]`
- Assuming page size is 128 words, then
- Each row (128 words) takes one page

If the process has fewer than 128 frames...

Program 1:

```
for(j=0;j<128;j++)
  for(i=0;i<128;i++)
    data[i][j] = 0;
```

Worst case:

$128 \times 128 = 16,384$ page faults

Program 2:

```
for(i=0;i<128;i++)
  for(j=0;j<128;j++)
    data[i][j] = 0;
```

Worst case:

128 page faults

Other Issues — I/O interlock

Sometimes it is necessary to lock pages in memory so that they are not paged out.

Example

- The OS
- I/O operation — the frame into which the I/O device was scheduled to write should not be replaced.
- New page that was just brought in — looks like the best candidate to be replaced because it was not accessed yet, nor was it modified.

Other Issues — I/O interlock

Case 1

Be sure the following sequence of events does not occur:

1. A process issues an I/O request, and then queueing for that I/O device
2. The CPU is given to other processes
3. These processes cause page faults
4. The waiting process' page is unluckily replaced
5. When its I/O request is served, the specific frame is now being used by another process

Other Issues — I/O interlock

Case 2

Another bad sequence of events:

1. A low-priority process faults
2. The paging system selects a replacement frame. Then, the necessary page is loaded into memory
3. The low-priority process is now ready to continue, and waiting in the ready queue
4. A high-priority process faults
5. The paging system looks for a replacement
   - 5.1 It sees a page that is in memory but not been referenced nor modified: perfect!
   - 5.2 It doesn't know the page is just brought in for the low-priority process

14.9 Segmentation

Two Views of A Virtual Address Space

One-dimensional

a linear array of bytes

Two-dimensional

a collection of variable-sized segments

User's View

- A program is a collection of segments
- A segment is a logical unit such as:

main program	procedure	function
method	object	local variables
global variables	common block	stack
symbol table	arrays	

Logical And Physical View of Segmentation

user space

physical memory space

Segmentation Architecture

- Logical address consists of a two tuple:

  `<segment-number, offset>`

- Segment table maps 2D virtual addresses into 1D physical addresses; each table entry has:
  - base contains the starting physical address where the segments reside in memory
  - limit specifies the length of the segment
- Segment-table base register (STBR) ☛ the segment table's location in memory
- Segment-table length register (STLR) indicates number of segments used by a program;

  segment number `s` is legal if `s < STLR`

Segmentation hardware

logical address space

	limit	base
0	1000	1400
1	400	6300
2	400	4300
3	1100	3200
4	1000	4700

segment table

physical memory

$$(2, 53) \Rightarrow 4300 + 53 = 4353$$
$$(3, 852) \Rightarrow 3200 + 852 = 4052$$
$$(0, 1222) \Rightarrow \text{Trap!}$$

Advantages of Segmentation

- Each segment can be
  - located independently
  - separately protected
  - grow independently
- Segments can be shared between processes

Problems with Segmentation

- Variable allocation
- Difficult to find holes in physical memory
- Must use one of non-trivial placement algorithm
  - first fit, best fit, worst fit
- External fragmentation

Linux prefers paging to segmentation

Because

- Segmentation and paging are somewhat redundant
- Memory management is simpler when all processes share the same set of linear addresses
- Maximum portability. RISC architectures in particular have limited support for segmentation

The Linux 2.6 uses segmentation only when required by the 80x86 architecture.

Case Study: The Intel Pentium

Segmentation With Paging

Logical address
(far pointer)
Segment
Selector
Index
0
p
Offset
Global Descriptor
Table (GDT)
...
Segment
Descriptor
...
...
GDTR
Linear address
space
...
...
Segment
... .
Linear addr.
... .
...
...
page
base
Linear Address
Dir
Table
Offset
Page
Directory
...
Entry
...
CR3
Page
Table
...
Entry
...
Physical
address space
...
... ...
Phy Addr.
... ...
...
Frame
Segmentation
Paging

Segmentation

Logical Address ⇒ Linear Address

Segment Selectors

A logical address consists of two parts:

segment selector	:	offset
16 bits		32 bits

Segment selector is an index into GDT/LDT

```
s – segment number
g – 0 (global)
    1 (local)
p – protection use
```

Segment Descriptor Tables

All the segments are organized in 2 tables:

GDT Global Descriptor Table

- shared by all processes
- GDTR stores address and size of the GDT

LDT Local Descriptor Table

- one process each
- LDTR stores address and size of the LDT

Segment descriptors are entries in either GDT or LDT, 8-byte long

Analogy

Process	$\Longleftrightarrow$	Process Descriptor(PCB)
File	$\Longleftrightarrow$	Inode
Segment	$\Longleftrightarrow$	Segment Descriptor

Segment Registers

The Intel Pentium has

- 6 segment registers, allowing 6 segments to be addressed at any one time by a process
  - Each segment register ☛ an entry in LDT/GDT
- 6 8-byte micro program registers to hold descriptors from either LDT or GDT
  - avoid having to read the descriptor from memory for every memory reference

```
   0          1          2          ...         9
┌──────────┬──────────┐──────────────────//──────────┐
│ Segment  selector │  Micro program register  │
└──────────┴──────────┘──────────────────//──────────┘
   Programmable          Non-programmable
```

Fast access to segment descriptors

An additional nonprogrammable register for each segment register

Segment registers hold segment selectors

- **cs** code segment register
  - **CPL** 2-bit, specifies the Current Privilege Level of the CPU

    - **00** - Kernel mode
    - **11** - User mode
- **ss** stack segment register
- **ds** data segment register
- **es/fs/gs** general purpose registers, may refer to arbitrary data segments

Example: A LDT entry for code segment

Base: Where the seg starts

G: Granularity flag
0 - seg size in bytes
1 - in 4096 bytes

D/B:
0 - 16-bit seg
1 - 32-bit seg

DPL: Descriptor Privilege Level. 0 or 3

AVL: ignored by Linux

Limit: 20 bit, $\Rightarrow 2^{20}$ in size

S: System flag
0 - system seg, e.g. LDT
1 - normal code/data seg

Type: seg type (cs/ds/tss)

P: Seg-Present flag
0 - not in memory
1 - in memory

The Four Main Linux Segments

Every process in Linux has these 4 segments

Segment	Base	G	Limit	S	Type	DPL	D/B	P
user code	`0x00000000`	1	`0xfffff`	1	10	3	1	1
user data	`0x00000000`	1	`0xfffff`	1	2	3	1	1
kernel code	`0x00000000`	1	`0xfffff`	1	10	0	1	1
kernel data	`0x00000000`	1	`0xfffff`	1	2	0	1	1

All linear addresses start at 0, end at 4G-1

- All processes share the same set of linear addresses
- Logical addresses coincide with linear addresses

Pentium Paging

Linear Address ⇒ Physical Address

Two page size in Pentium:

4K: 2-level paging

4M: 1-level paging

Paging In Linux

4-level paging for both 32-bit and 64-bit

4-level paging for both 32-bit and 64-bit

- 64-bit: four-level paging
  1. Page Global Directory
  2. Page Upper Directory
  3. Page Middle Directory
  4. Page Table
- 32-bit: two-level paging
  1. Page Global Directory
  2. Page Upper Directory — 0 bits; 1 entry
  3. Page Middle Directory — 0 bits; 1 entry
  4. Page Table

The same code can work on 32-bit and 64-bit architectures

Arch	Page size	Address bits	Paging levels	Address splitting
x86	4KB(12bits)	32	2	$10+0+0+10+12$
x86-PAE	4KB(12bits)	32	3	$2+0+9+\ \ 9+12$
x86-64	4KB(12bits)	48	4	$9+9+9+\ \ 9+12$

References

Wikipedia. *Memory management — Wikipedia, The Free Encyclopedia*. 2015. http://en.wikipedia.org/w/index.php?title=Memory%5C_management&oldid=647917932.

Wikipedia. *Virtual memory — Wikipedia, The Free Encyclopedia*. 2015. http://en.wikipedia.org/w/index.php?title=Virtual%5C_memory&oldid=644430956.

Part IV

File Systems

15 File System Structure

Long-term Information Storage Requirements

- Must store large amounts of data
- Information stored must survive the termination of the process using it
- Multiple processes must be able to access the information concurrently

File-System Structure

File-system design addressing two problems:

1. defining how the FS should look to the user
   - defining a file and its attributes
   - the operations allowed on a file
   - directory structure
2. creating algorithms and data structures to map the logical FS onto the physical disk

File-System — A Layered Design

APPs
⇓
Logical FS
⇓
File-org module
⇓
Basic FS
⇓
I/O ctrl
⇓
Devices

- logical file system — manages metadata information
  - maintains all of the file-system structure (directory structure, FCB)
  - responsible for protection and security
- file-organization module
  - logical block address $\xrightarrow{\textit{translate}}$ physical block address
  - keeps track of free blocks
- basic file system issues generic commands to device driver, e.g
  - "`read drive 1, cylinder 72, track 2, sector 10`"
- I/O Control — device drivers, and INT handlers
  - device driver: high-level commands $\xrightarrow{\textit{translate}}$ hardware-specific instructions

The Operating Structure

APPs
⇓
Logical FS
⇓
File-org module
⇓
Basic FS
⇓
I/O ctrl
⇓
Devices

Example — To create a file

1. APP calls `creat()`
2. Logical FS
   - 2.1 allocates a new FCB
   - 2.2 updates the in-mem dir structure
   - 2.3 writes it back to disk
   - 2.4 calls the file-org module
3. file-organization module
   - 3.1 allocates blocks for storing the file's data
   - 3.2 maps the directory I/O into disk-block numbers

Benefit of layered design

The I/O control and sometimes the basic file system code can be used by multiple file systems.

16 Files

File

A Logical View Of Information Storage

User's view

A file is the smallest storage unit on disk.

- Data cannot be written to disk unless they are within a file

UNIX view

Each file is a sequence of 8-bit bytes

- It's up to the application program to interpret this byte stream.

File

What Is Stored In A File?

Source code, object files, executable files, shell scripts, PostScript...

Different type of files have different structure

- UNIX looks at contents to determine type

  Shell scripts start with "`#!`"

  PDF start with "`%PDF...`"

  Executables start with magic number
- Windows uses file naming conventions

  executables end with "`.exe`" and "`.com`"

  MS-Word end with "`.doc`"

  MS-Excel end with "`.xls`"

File Naming

Vary from system to system

- Name length?
- Characters? Digits? Special characters?
- Extension?
- Case sensitive?

File Types

Regular files: ASCII, binary

Directories: Maintaining the structure of the FS

In UNIX, everything is a file

Character special files: I/O related, such as terminals, printers ...

Block special files: Devices that can contain file systems, i.e. disks

- **disks** — logically, linear collections of blocks; disk driver translates them into physical block addresses

Binary files

File Attributes — Metadata

- Name - only information kept in human-readable form
- Identifier - unique tag (number) identifies file within file system
- Type - needed for systems that support different types
- Location - pointer to file location on device
- Size - current file size
- Protection - controls who can do reading, writing, executing
- Time, date, and user identification - data for protection, security, and usage monitoring

File Operations

POSIX file system calls

```
creat(name, mode)
open(name, flags)
close(fd)
link(oldname, newname)
unlink(name)
truncate(name, size)
ftruncate(fd, size)
stat(name, buffer)
fstat(fd, buffer)
```

```
read(fd, buffer, byte_count)
write(fd, buffer, byte_count)
lseek(fd, offset, whence)
chown(name, owner, group
fchown(fd, owner, group)
chmod(name, mode
fchmod(fd, mode)
utimes(name, times)
```

write()

```c
#include <unistd.h>

int main(void)
{
  write(1, "Hello, world!\n", 14);

  return 0;
}
```

```
$ man 2 write
$ man 3 write
```

read()

```c
#include <unistd.h>

int main(void)
{
  char buffer[10];

  read(0, buffer, 10);

  write(1, buffer, 10);

  return 0;
}
```

```
$ man 2 read
$ man 3 read
```

- No need to `open()` `STDIN`, `STDOUT`, and `STDERR`

cp

```
#define BUF_SIZE 4096
#define OUTPUT_MODE 0700

int main(int argc, char *argv[])
{
  int in, out, rbytes, wbytes;
  char buf[BUF_SIZE];

  if (argc != 3) exit(1);

  if ( (in = open(argv[1], O_RDONLY)) < 0 ) exit(2);

  if ( (out = creat(argv[2], OUTPUT_MODE)) < 0 ) exit(3);

  while (1) { /* Copy loop */
    if ( (rbytes = read(in, buf, BUF_SIZE)) <= 0 ) break;
    if ( (wbytes = write(out, buf, rbytes)) <= 0 ) exit(4);
  }

  close(in); close(out);
  if (rbytes == 0) exit(0); /* no error on last read */
  else exit(5);             /* error on last read */
}
```

`stdio` — The Standard I/O Library

System calls: `open()`, `read()`, `write()`, `close()`...

Library functions: `fopen()`, `fread()`, `fwrite`, `fclose()`...

Avoid calling syscalls directly as much as you can

- Portability
- Buffered I/O

open() vs. fopen()

open()

```c
#include <unistd.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>

int main()
{
  char c;
  int in;
  in = open("/tmp/1m.test", O_RDONLY);

  while (read(in, &c, 1) == 1);

  return 0;
}
```

fopen() — Buffered I/O

```c
#include <stdio.h>

int main(void)
{
  FILE *stream;

  stream = fopen("/tmp/1m.test", "r");

  while ( fgetc(stream) != EOF );

  fclose(stream);

  return 0;
}
```

```
$ strace -c ./open
```

```
$ strace -c ./fopen
```

```
$ dd if=/dev/zero of=/tmp/1m.test bs=1k count=1024
```

cp — With stdio

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

int main(int argc, char *argv[])
{
        FILE *in, *out;
        int c=0;

        if (argc != 3) exit(1);

        in = fopen(argv[1], "r");
        out = fopen(argv[2], "w");

        while ( (c = fgetc(in)) != EOF )
                fputc(c, out);

        return 0;
}
```

Try `fread()/fwrite()` instead.

open()

fd open(pathname, flags)

A per-process open-file table is kept in the OS

- upon a successful `open()` syscall, a new entry is added into this table
- indexed by file descriptor (fd)

To see files opened by a process, e.g. `init`

```
$ lsof -p 1
```

Why open() is needed?

To avoid constant searching

- Without `open()`, every file operation involves searching the directory for the file.

17 Directories

Directories

Single-Level Directory Systems

All files are contained in the same directory

- contains 4 files
- owned by 3 different people, A, B, and C

Limitations

- name collision
- file searching

Often used on simple embedded devices, such as telephone, digital cameras...

Directories

Two-level Directory Systems

A separate directory for each user

Limitation: hard to access others files

Directories

Hierarchical Directory Systems

Directories

Path Names

Directories

Directory Operations

```
opendir()   mkdir()   rename()   link()
closedir()  rmdir()   readdir()   unlink()
…
```

ls

```c
#include <sys/types.h>
#include <dirent.h>
#include <stddef.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
  DIR *dp;
  struct dirent *entry;

  dp = opendir(argv[1]);

  while ( (entry = readdir(dp)) != NULL ){
    printf("%s\n", entry->d_name);
  }

  closedir(dp);

  return 0;
}
```

The real `ls.c`?

116 A4 pages
5308 lines

Do one thing, and do it really well.

```
$ apt source coreutils
```

mkdir(), chdir(), rmdir(), getcwd()

```
1   #include <sys/stat.h>
2   #include <sys/types.h>
3   #include <unistd.h>
4   #include <stdio.h>
5
6   int main(int argc, char *argv[])
7   {
8     char s[100];
9     if ( mkdir(argv[1], S_IRUSR|S_IXUSR) == 0 )
10      chdir(argv[1]);
11    printf("PWD = %s\n", getcwd(s,100));
12    rmdir(argv[1]);
13    return 0;
14  }
```

18 File System Implementation

18.1 Basic Structures

File System Implementation

A typical file system layout

```
|<------------------------- Entire disk ------------------------------>|
|-----------------------------------------------------------------------|
| MBR | Partition 1 | Partition 2 | Partition 3 | Partition 4 |
|-----------------------------------------------------------------------|

 ____________________________________/                 \_____________
/                                                                      \
                                                                    //
| Boot Ctrl Blk | Volume Ctrl Blk | Dir Structure | Files |
|  (MBR copy)   |   (Super Blk)   | (inodes, root dir) | dirs |
                                                                    //
```

```
|<------------- Master Boot Record (512 Bytes) ------------->|
|-----//-------------------------//-------------------------|
| code area | disk-sig | null | partition table | MBR-sig |
|    440    |    4     |  2   |     16x4=64     | 0xAA55  |
|-----//-------------------------//-------------------------|
0          439        443    445              509       511
```

On-Disk Information Structure

Boot control block a MBR copy

- UFS: Boot block
- NTFS: Partition boot sector

Volume control block Contains volume details

number of blocks	size of blocks
free-block count	free-block pointers
free FCB count	free FCB pointers

- UFS: Superblock
- NTFS: Master File Table

Directory structure Organizes the files FCB, File control block, contains file details (metadata).

- UFS: I-node
- NTFS: Stored in MFT using a relational database structure, with one row per file

Each File-System Has a Superblock

Superblock

Keeps information about the file system

- Type — ext2, ext3, ext4...
- Size
- Status — how it's mounted, free blocks, free inodes, ...
- Information about other metadata structures

```
$ sudo dumpe2fs /dev/sda1 | less
```

18.2 Implementing Files

Implementing Files

Contiguous Allocation

File Allocation Table

File Name	Start Block	Length
File A	2	3
File B	9	5
File C	18	8
File D	30	2
File E	26	3

- simple;
- good for read only;
- fragmentation

Linked List (Chained) Allocation A pointer in each disk block

File Allocation Table

File Name	Start Block	Length
•••	•••	•••
File B	1	5
•••	•••	•••

- no waste block; access;
- slow random
- not 2^n

Linked List (Chained) Allocation Though there is no external fragmentation, consolidation is still preferred.

File Allocation Table

File Name	Start Block	Length
•••	•••	•••
File B	0	5
•••	•••	•••

FAT: Linked list allocation with a table in RAM

- Taking the pointer out of each disk block, and putting it into a table in memory
- fast random access (chain is in RAM)
- is 2^n
- the entire table must be in RAM

$$disk \nearrow \Rightarrow FAT \nearrow \Rightarrow RAM_{used} \nearrow$$

Physical block		
0		
1		
2	10	
3	11	
4	7	<- File A starts here
5		
6	3	<- File B starts here
7	2	
8		
9		
10	12	
11	14	
12	−1	
13		
14	−1	
15		<- Unused block

Indexed Allocation

I-node A data structure for each file. An i-node is in memory *only if* the file is open

$$files_{opened} \nearrow \Rightarrow RAM_{used} \nearrow$$

I-node — FCB in UNIX

File type	Description
0	Unknown
1	Regular file
2	Directory
3	Character device
4	Block device
5	Named pipe
6	Socket
7	Symbolic link

Mode: 9-bit pattern

Inode Quiz

Given: block size is 1KB; pointer size is 4B

Addressing: byte offset 9000; byte offset 350,000

UNIX In-Core Data Structure

mount table — Info about each mounted FS

directory-structure cache — Dir-info of recently accessed dirs

inode table — An in-core version of the on-disk inode table

file table

- global
- keeps inode of each open file
- keeps track of
  - how many processes are associated with each open file
  - where the next read and write will start
  - access rights

user file descriptor table

- per process
- identifies all open files for a process

UNIX In-Core Data Structure

`open()/creat()`

1. add entry in each table
2. returns a file descriptor — an index into the user file descriptor table

The Tables

Two levels of internal tables in the OS

A per-process table tracks all files that a process has open. Stores

- the current-file-position pointer (not really)
- access rights
- more...

a.k.a file descriptor table

A system-wide table keeps process-independent information, such as

- the location of the file on disk
- access dates
- file size
- file open count — the number of processes opening this file

Per-process FDT
Process 1
...
Position pointer
Access rights
...
...
Process 2
Position pointer
Access rights
...
...
System-wide
open-file table
...
...
Location on disk
R/W
Access dates
File size
Pointer to inode
File-open count
...
...

A process executes the following code:

```
fd1 = open("/etc/passwd", O_RDONLY);
fd2 = open("local", O_RDWR);
fd3 = open("/etc/passwd", O_WRONLY);
```

One more process B:

```
fd1 = open("/etc/passwd", O_RDONLY);
fd2 = open("private", O_RDONLY);
```

Why File Table?

To allow a parent and child to share a file position, but to provide unrelated processes with their own values.

Why File Table?

Where To Put File Position Info?

Inode table? No. Multiple processes can open the same file. Each one has its own file position.

User file descriptor table? No. Trouble in file sharing.

Example

```
1  #!/bin/bash
2
3  echo hello
4  echo world
```

```
$ ./hello.sh > A
```

? Where should the "`world`" be?

18.3 Implementing Directories

Implementing Directories

(a) A simple directory (Windows)

- fixed size entries
- disk addresses and attributes in directory entry

(b) Directory in which each entry just refers to an i-node (UNIX)

Directory entry in `glibc`

```c
struct dirent {
  ino_t          d_ino;       /* Inode number */
  off_t          d_off;       /* Not an offset; see below */
  unsigned short d_reclen;    /* Length of this record */
  unsigned char  d_type;      /* Type of file; not supported
                                 by all filesystem types */
  char           d_name[256]; /* Null-terminated filename */
};
```

```
$ man readdir
$ view /usr/include/x86_64-linux-gnu/bits/dirent.h
```

How Long A File Name Can Be?

UNIX Treats a Directory as a File

.	2
..	2
bin	11116545
boot	2
cdrom	12
dev	3
⋮	⋮

The steps in looking up `/usr/ast/mbox`

Root directory

1	.
1	..
4	bin
7	dev
14	lib
9	etc
6	usr
8	tmp

Looking up usr yields i-node 6

I-node 6 is for /usr

Mode size times
132

I-node 6 says that /usr is in block 132

Block 132 is /usr directory

6	•
1	••
19	dick
30	erik
51	jim
26	ast
45	bal

/usr/ast is i-node 26

I-node 26 is for /usr/ast

Mode size times
406

I-node 26 says that /usr/ast is in block 406

Block 406 is /usr/ast directory

26	•
6	••
64	grants
92	books
60	mbox
81	minix
17	src

/usr/ast/mbox is i-node 60

18.4 Shared Files

File Sharing

Multiple Users

User IDs identify users, allowing permissions and protections to be per-user

Group IDs allow users to be in groups, permitting group access rights

Example: 9-bit pattern

`rwxr-x---` means:

user	group	other
rwx	r-x	---
111	1-1	000
7	5	0

File Sharing

Remote File Systems

Networking — allows file system access between systems

- Manually via programs like FTP
- Automatically, seamlessly using distributed file systems
- Semi automatically, via the world wide web

C/S model — allows clients to *mount* remote file systems from servers

- NFS — standard UNIX client-server file sharing protocol
- CIFS — standard Windows protocol
- Standard system calls are translated into remote calls

Distributed Information Systems (distributed naming services)

- such as LDAP, DNS, NIS, Active Directory implement unified access to information needed for remote computing

File Sharing

Protection

- File owner/creator should be able to control:
  - what can be done
  - by whom
- Types of access
  - Read
  - Write
  - Execute
  - Append
  - Delete
  - List

Shared Files

Hard Links vs. Soft Links

Hard Links

Hard links ☛ the same inode

```
directory entry in /dirA

    inode    name
  ┌────────┬────────┐
  │ 12345  │ name1  │
  └────────┴────────┘

 inode 12345
 ┌──────────┐
 │          │
 │    1     │
 │          │
 │  23567   │──────>  block 23567
 │          │         ┌──────────────┐
 │          │         │"text in      │
 └──────────┘         │ the file"    │
                      └──────────────┘
```

```
dir entry in /dirA          dir entry in /dirB

    inode    name               inode    name
  ┌────────┬────────┐         ┌────────┬────────┐
  │ 12345  │ name1  │         │ 12345  │ name2  │
  └────────┴────────┘         └────────┴────────┘

 inode 12345                  <
 ┌──────────┐
 │          │
 │    1     │                 block 23567
 │          │                 ┌──────────────┐
 │          │                 │"text in      │
 │  23567   │──────>          │ the file"    │
 │          │                 └──────────────┘
 └──────────┘
```

Drawback

C's directory
Owner = C
Count = 1
(a)
B's directory
C's directory
Owner = C
Count = 2
(b)
B's directory
Owner = C
Count = 1
(c)

Symbolic Links

A symbolic link has its own inode ☛ a directory entry

link(), unlink(), symlink()

```
1   #include <unistd.h>
2   #include <stdio.h>
3
4   int main(int argc, char *argv[])
5   {
6     link(argv[1], argv[2]);
7     perror(argv[0]);
8     return 0;
9   }
10
11  /* symlink(argv[1], argv[2]); */
12  /* unlink(argv[1]); */
```

18.5 Disk Space Management

Disk Space Management

Statistics

Length	VU 1984	VU 2005	Web
1	1.79	1.38	6.67
2	1.88	1.53	7.67
4	2.01	1.65	8.33
8	2.31	1.80	11.30
16	3.32	2.15	11.46
32	5.13	3.15	12.33
64	8.71	4.98	26.10
128	14.73	8.03	28.49
256	23.09	13.29	32.10
512	34.44	20.62	39.94
1 KB	48.05	30.91	47.82
2 KB	60.87	46.09	59.44
4 KB	75.31	59.13	70.64
8 KB	84.97	69.96	79.69

Length	VU 1984	VU 2005	Web
16 KB	92.53	78.92	86.79
32 KB	97.21	85.87	91.65
64 KB	99.18	90.84	94.80
128 KB	99.84	93.73	96.93
256 KB	99.96	96.12	98.48
512 KB	100.00	97.73	98.99
1 MB	100.00	98.87	99.62
2 MB	100.00	99.44	99.80
4 MB	100.00	99.71	99.87
8 MB	100.00	99.86	99.94
16 MB	100.00	99.94	99.97
32 MB	100.00	99.97	99.99
64 MB	100.00	99.99	99.99
128 MB	100.00	99.99	100.00

- Block size is chosen while creating the FS
- Disk I/O performance is conflict with space utilization
  - smaller block size $\Rightarrow$ better space utilization
  - larger block size $\Rightarrow$ better disk I/O performance

```
$ dumpe2fs /dev/sda1 | grep "Block size"
```

Keeping Track of Free Blocks

1. Linked List

2. Bit map (n blocks)

```
 0 1 2 3 4 5 6 7 8  .. n-1
                   //
| | | | | | | | |  //  |
0 0 1 0 1 1 1 0 1  ..  0
| | | | | | | | |  //  |
                   //
```

$$bit[i] = \begin{cases} 0 \Rightarrow block[i] \text{ is free} \\ 1 \Rightarrow block[i] \text{ is occupied} \end{cases}$$

Journaling File Systems

Operations required to remove a file in UNIX:

1. Remove the file from its directory
   - set inode number to 0
2. Release the i-node to the pool of free i-nodes
   - clear the bit in inode bitmap
3. Return all the disk blocks to the pool of free disk blocks
   - clear the bits in block bitmap

What if crash occurs between 1 and 2, or between 2 and 3?

Journaling File Systems

Keep a log of what the file system is going to do before it does it

- so that if the system crashes before it can do its planned work, upon rebooting the system can look in the log to see what was going on at the time of the crash and finish the job.
- NTFS, EXT3, and ReiserFS use journaling among others

19 Ext2 File System

19.1 Ext2 File System Layout

Ext2 File System

Physical Layout

```
                                                        //
 ┌──────────────┬──────────────────────────────────────┬──────────────────────┬──────┬──────────────────────┐
 │ Boot Block   │ Block Group 0                        │ Block Group 1        │      │ Block Group n        │
 └──────────────┴──────────────────────────────────────┴──────────────────────┴──────┴──────────────────────┘
                                                        //
  ____________________________/                    \_______________
 /                                                                 \

 ┌──────────┬──────────────┬──────────────┬──────────┬──────────┬──────────┐
 │ Super    │ Group        │ Data Block   │ inode    │ inode    │ Data     │
 │ Block    │ Descriptors  │ Bitmap       │ Bitmap   │ Table    │ Blocks   │
 └──────────┴──────────────┴──────────────┴──────────┴──────────┴──────────┘

   1 blk       n blks          1 blk        1 blk     n blks     n blks
```

19.2 Ext2 Block groups

Ext2 Block groups

The partition is divided into Block Groups

- Block groups are same size — easy locating
- Kernel tries to keep a file's data blocks in the same block group — reduce fragmentation
- Backup critical info in each block group
- The Ext2 inodes for each block group are kept in the inode table
- The inode-bitmap keeps track of allocated and unallocated inodes

Group descriptor

- Each block group has a group descriptor
- All the group descriptors together make the group descriptor table
- The table is stored along with the superblock
- Block Bitmap: tracks free blocks
- Inode Bitmap: tracks free inodes
- Inode Table: all inodes in this block group
- Free blocks count, Free Inodes count, Used dir count } counters

```
# dumpe2fs /dev/sda1
```

Maths

Given $\begin{array}{l} \textit{block size} = 4K \\ \textit{block bitmap} = 1\,\textit{blk} \end{array}$, then

$$\textit{blocks per group} = 8\,\textit{bits} \times 4K = 32K$$

How large is a group?

$$\textit{group size} = 32K \times 4K = 128\,M$$

How many block groups are there?

$$\approx \frac{\textit{partition size}}{\textit{group size}} = \frac{\textit{partition size}}{128M}$$

How many files can I have in max?

$$\approx \frac{\textit{partition size}}{\textit{block size}} = \frac{\textit{partition size}}{4K}$$

Ext2 Block Allocation Policies

19.3 Ext2 Inode

Ext2 inode

Tyep
Mode
UID
GID
timestamps (3)
size block count
...
direct blocks
single indirect
double indirect
triple indirect
data
data
data
data
PTR
data
data
PTR
PTR
data
data
PTR
data

Ext2 inode

Mode: holds two pieces of information

1. Is it a {*file*|*dir*|*sym-link*|*blk-dev*|*char-dev*|*FIFO*}?
2. Permissions

Owner info: Owners' ID of this file or directory

Size: The size of the file in bytes

Timestamps: Accessed, created, last modified time

Datablocks: 15 pointers to data blocks ($12 + S + D + T$)

Max File Size

Given:

$$\begin{cases} \textit{block size} = 4K \\ \textit{pointer size} = 4B \end{cases},$$

We get:

$$\textit{Max File Size} = \textit{number of pointers} \times \textit{block size}$$

$$= (\overbrace{\underbrace{12}_{\textit{direct}} + \underbrace{1K}_{1-\textit{indirect}} + \underbrace{1K \times 1K}_{2-\textit{indirect}} + \underbrace{1K \times 1K \times 1K}_{3-\textit{indirect}}}^{\textit{number of pointers}}) \times 4K$$

$$= 48K + 4M + 4G + 4T$$

19.4 Ext2 Superblock

Ext2 Superblock

Magic Number: `0xEF53`

Revision Level: determines what new features are available

Mount Count and Maximum Mount Count: determines if the system should be fully checked

Block Group Number: indicates the block group holding this superblock

Block Size: usually $4K$

Blocks per Group: $8 \textit{bits} \times \textit{block size}$

Free Blocks: System-wide free blocks

Free Inodes: System-wide free inodes

First Inode: First inode number in the file system

See more:

```
# dumpe2fs /dev/sda1
```

Ext2 File Types

File type	Description
0	Unknown
1	Regular file
2	Directory
3	Character device
4	Block device
5	Named pipe
6	Socket
7	Symbolic link

Device file, pipe, and socket: No data blocks are required. All info is stored in the inode

Fast symbolic link: Short path name (< 60 *chars*) needs no data block. Can be stored in the 15 pointer fields

19.5 Ext2 Directory

Ext2 Directories

- Directories are special files
- "." and ". ." first
- Padding to $4 \times$
- inode number is 0 — deleted file

20 Vitural File Systems

Many different FS are in use

Windows

uses drive letter (`C:`, `D:`, ...) to identify each FS

UNIX

integrates multiple FS into a single structure

- From user's view, there is only one FS hierarchy

```
$ man fs
```

```
$ cp /floppy/TEST /tmp/test
```

```
inf = open("/floppy/TEST", O_RDONLY, 0);
outf = open("/tmp/test",
            O_WRONLY|O_CREAT|O_TRUNC, 0600);
do{
    i = read(inf, buf, 4096);
    write(outf, buf, i);
} while (i);
close(outf);
close(inf);
```

```
ret = write(fd, buf, len);
```

Virtural File Systems

Put common parts of all FS in a separate layer

- It's a layer in the kernel
- It's a common interface to several kinds of file systems
- It calls the underlying concrete FS to actual manage the data

Virtual File System

- Manages kernel level file abstractions in one format for all file systems
- Receives system call requests from user level (e.g. `write`, `open`, `stat`, `link`)
- Interacts with a specific file system based on mount point traversal
- Receives requests from other parts of the kernel, mostly from memory management

Real File Systems

- managing file & directory data
- managing meta-data: timestamps, owners, protection, etc.
- $\boxed{\text{disk data, NFS data...}} \xleftrightarrow{\textit{translate}} \boxed{\text{VFS data}}$

File System Mounting

A FS must be mounted before it can be used

Mount — The file system is registered with the VFS

- The superblock is read into the VFS superblock
- The table of addresses of functions the VFS requires is read into the VFS superblock
- The FS' topology info is mapped onto the VFS superblock data structure

The VFS keeps a list of the mounted file systems together with their superblocks

The VFS superblock contains:

- Device, blocksize
- Pointer to the root inode
- Pointer to a set of superblock routines
- Pointer to `file_system_type` data structure
- more...

V-node

- Every file/directory in the VFS has a VFS inode, kept in the VFS inode cache
- The real FS builds the VFS inode from its own info

Like the EXT2 inodes, the VFS inodes describe

- files and directories within the system
- the contents and topology of the Virtual File System

VFS Operation

`read()`

Linux VFS

The Common File Model

All other filesystems must map their own concepts into the common file model

For example, FAT filesystems do not have inodes.

- The main components of the common file model are

  superblock – information about mounted filesystem
  inode – information about a specific file
  file – information about an open file
  dentry – information about directory entry

- Geared toward Unix FS

The Superblock Object

- is implemented by each FS and is used to store information describing that specific FS
- usually corresponds to the filesystem superblock or the filesystem control block
- Filesystems that are not disk-based (such as `sysfs`, `proc`) generate the superblock on-the-fly and store it in memory
- `struct super_block` in `<linux/fs.h>`
- `s_op` in `struct super_block` ☛ `struct super_operations` — the superblock operations table
  - Each item in this table is a pointer to a function that operates on a superblock object

The Inode Object

- For Unix-style filesystems, this information is simply read from the on-disk inode
- For others, the inode object is constructed in memory in whatever manner is applicable to the filesystem
- `struct inode` in `<linux/fs.h>`
- An `inode` represents each file on a FS, but the `inode object` is constructed in memory only as files are accessed
  - includes special files, such as device files or pipes
- `i_op` ☛ `struct inode_operations`

The Dentry Object

- components in a path
- makes path name lookup easier
- `struct dentry` in `<linux/dcache.h>`
- created on-the-fly from a string representation of a path name

Dentry State

- used
- unused
- negative

Dentry Cache

consists of three parts:

1. Lists of "used" dentries
2. A doubly linked "least recently used" list of unused and negative dentry objects
3. A hash table and hashing function used to quickly resolve a given path into the associated dentry object

The File Object

- is the in-memory representation of an open file
- `open()` ⇒ create; `close()` ⇒ destroy
- there can be multiple file objects in existence for the same file
  - Because multiple processes can open and manipulate a file at the same time
- `struct file` in `<linux/fs.h>`

disk
file
Superblock
object
i_sb
inode
object
d_inode
Process 1
fd
File object
f_dentry
Process 2
File object
dentry
object
dentry
object
Process 3
File object

References

- Wikipedia. *File system — Wikipedia, The Free Encyclopedia*. 2015. http://en.wikipedia.org/w/index.php?title=File%5C_system&oldid=646603121.
- Wikipedia. *Computer file — Wikipedia, The Free Encyclopedia*. 2015. http://en.wikipedia.org/w/index.php?title=Computer%5C_file&oldid=647724614.
- Wikipedia. *Inode — Wikipedia, The Free Encyclopedia*. 2015. http://en.wikipedia.org/w/index.php?title=Inode&oldid=647736522.
- Wikipedia. *Ext2 — Wikipedia, The Free Encyclopedia*. 2015. http://en.wikipedia.org/w/index.php?title=Ext2&oldid=642312602.
- Wikipedia. *Virtual file system — Wikipedia, The Free Encyclopedia*. 2014. http://en.wikipedia.org/w/index.php?title=Virtual%5C_file%5C_system&oldid=640354992.