Week 08

File System Operations

1/35

Unix presents a uniform interface to file system objects

- functions/syscalls manipulate objects as a *stream of bytes*
- accessed via a *file descriptor* (index into a system table)

Some common operations:

- `open()`, `close()`, `read()`, `write()` ... defined already
- `lseek()` ... move to a specified offset within a file
- `stat()` ... get meta-data about a file system object
- `mkdir()` ... create a new directory
- `fsync()` ... synchronise file data in memory with data on disk
- `mount()` ... place a filesystem on a device

... File System Operations

2/35

Functions from `stdio.h` tend to be `char`-oriented

File-descriptor-based system calls deal with byte sequences

- bytes can be interpreted as `char`, `int`, `struct`, etc
- so, many kinds of objects can be `read()` or `write()` **

Allows programmers to manipulate files of data items, e.g.

- list of `double` values read from sensor device
- collection of `Student` records

** you cannot save/restore pointer values using `write()`/`read()`

- because they refer to memory addresses within a process instance
- and a different process instance might already have used those addresses

... File System Operations

3/35

Files of *records* can be produced by

- either, `write()`ing chunks of bytes from `struct` objects
- or, printing formatted text representation of `struct` data

The latter approach is a form of *serialisation*

For the `write()` approach:

- no need to worry about formatting issues
- writes entire structure, even if string buffers half empty
- can `lseek()` to i^{th} struct via *i*sizeof(StructType)*

For the printing approach:

- produces files that are human-readable
- only uses as many bytes as required from string buffers
- can access structures only sequentially (unless using padding)

... File System Operations

4/35

Example of `write()`ing records vs `printf()`ing records

```
typedef struct _student {
    int id; char name[99]; float wam;
```

```
} Student;
int infd, outfd; // file descriptors
FILE *inf, *outf; // file pointers

Student stu;  ... set values in stu.id, etc ...

write(outfd, &stu, sizeof(struct _student));
  vs
fprintf(outf, "%d:%s:%f\n",
             stu.id, stu.name, stu.wam);

read(infd, &stu, sizeof(Student));
  vs
fscanf(inf, "%d:[^:]:%f\n", // maybe?
            &(stu.id), &(stu.name), &(stu.wam));
```

Exercise 1: Files of Structs

5/35

Consider a new `Student` struct ...

```
typedef struct _student {
  int id; char given[50]; char family[50]; int prog; float wam;
} Student;
```

Write a program to ...

- read in data about one student
- append the data to a file of students

Write a program to ...

- scan the file of students and print data for each one

Write two versions of each program ...

- one using the `write()/read()` approach
- one using the `printf()/scanf()` approach

... File System Operations

6/35

`off_t lseek(int` ***FileDesc*****`, off_t`** ***Offset*****`, int`** ***Whence*****`)`**

- set the "current position" of the *FileDesc*
- *Offset* is in units of bytes, and can be negative
- *Whence* can be one of ...
  - `SEEK_SET` ... set file position to *Offset* from start of file
  - `SEEK_CUR` ... set file position to *Offset* from current position
  - `SEEK_END` ... set file position to *Offset* from end of file
- seeking beyond end of file leaves a gap which reads as 0's
- seeking back beyond start of file sets position to start of file

Example: `lseek(fd, 0, SEEK_END);` (move to end of file)

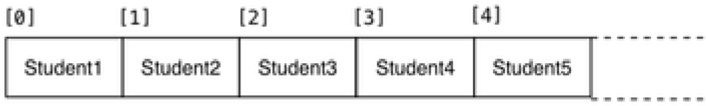
Exercise 2: Seeking to a Record

7/35

Create a new program that shows one particular `Student` record

- one parameter is the name of the file containing student data
- the other parameter is the index of the record to be displayed

```
./sstu Students 2
```



displays `Student3`

... File System Operations

8/35

`int stat(char *`*FileName*`, struct stat *`*StatBuf*`)`

- stores meta-data associated with *FileName* into *StatBuf*
- information includes
  - inode number, file type + access mode, owner, group
  - size in bytes, storage block size, allocated blocks
  - time of last access/modification/status-change
- returns -1 and sets `errno` if meta-data not accessible

`int fstat(int `*FileDesc*`, struct stat *`*StatBuf*`)`

- same as `stat()` but gets data via an open file descriptor

`int stat(char *`*FileName*`, struct stat *`*StatBuf*`)`

- same as `stat()` but doesn't follow symbolic links

... File System Operations

9/35

File system *links* allow multiple paths to access the same data

Hard links

- multiple directory entries referencing the same inode
- the two entries must be on the same filesystem

Symbolic links (symlinks)

- a file containing the path name of another file
- opening the symlink opens the file being referenced

Example:

```
-rw-r----- 2 cs1521 46 Sep 10 22:28 fileA
-rw-r----- 2 cs1521 46 Sep 10 22:28 fileB
lrwxrwxrwx 1 cs1521  5 Sep 10 22:29 fileC -> fileA
```

... File System Operations

10/35

File `stat` structure:

```
struct stat {
   dev_t     st_dev;     // ID of device containing file
   ino_t     st_ino;     // inode number
   mode_t    st_mode;    // file type + permissions
   nlink_t   st_nlink;   // number of hard links
   uid_t     st_uid;     // user ID of owner
   gid_t     st_gid;     // group ID of owner
   dev_t     st_rdev;    // device ID (if special file)
   off_t     st_size;    // total size, in bytes
   blksize_t st_blksize; // blocksize for file system I/O
   blkcnt_t  st_blocks;  // number of 512B blocks allocated
   time_t    st_atime;   // time of last access
   time_t    st_mtime;   // time of last modification
```

```
    time_t    st_ctime;    // time of last status change
};
```

... File System Operations

The `st_mode` is a bit-string containing some of:

```
S_IFLNK    0120000   symbolic link
S_IFREG    0100000   regular file
S_IFBLK    0060000   block device
S_IFDIR    0040000   directory
S_IFCHR    0020000   character device
S_IFIFO    0010000   FIFO

S_IRUSR    0000400   owner has read permission
S_IWUSR    0000200   owner has write permission
S_IXUSR    0000100   owner has execute permission

S_IRGRP    0000040   group has read permission
S_IWGRP    0000020   group has write permission
S_IXGRP    0000010   group has execute permission

S_IROTH    0000004   others have read permission
S_IWOTH    0000002   others have write permission
S_IXOTH    0000001   others have execute permission
```

... File System Operations

`int mkdir(char *`*PathName*`, mode_t` *Mode*`)`

- create a new directory called *PathName* with mode *Mode*
- if *PathName* is e.g. `a/b/c/d`
  - all of the directories `a`, `b` and `c` must exist
  - directory `c` must be writeable to the caller
  - directory `d` must not already exist
- the new directory contains two initial entries
  - **`.`** is a reference to itself
  - **`..`** is a reference to its parent directory
- returns 0 if successful, returns -1 and sets `errno` otherwise

Example: `mkdir("newDir", 0755);`

... File System Operations

`int fsync(int` *FileDesc*`)`

- ensure that data associated with *FileDesc* is written to storage

Unix/Linux makes heavy use of buffering

- data "written" to a file is initially stored in memory buffers
- eventually, it makes its way onto permanent storage device
- `fsync()` forces this to happen *now*

Writing to permanent storage is typically an expensive operation

- `fsync()` is normally called just once at process exit

Note also: `fflush()` forces stdio buffers to be copied to kernel buffers

... File System Operations

`int mount(char *`*Source*`, char *`*Target*`, char *`*FileSysType*`,`
`unsigned long` *Flags*`, void *`data`)`

- file systems normally exist on permanent storage devices
- `mount` attaches a file system to a specific location in the file hierarchy
- *Source* is often a storage device (e.g. `/dev/disk`)
- *Source* contains a file system (inode table, data chunks)
- *Target* (aka *mount point*) is a path in the file hierarchy
- *FileSysType* specifies a particular layout/drivers
- *Flags* specify various properties of the filesys (e.g. read-only)

Example: `mount("/dev/disk5","/usr","ext3",MS_RDONLY,...)`

(use `disk5` to hold the `/usr` file system as read-only ext3-type)

File System Summary

15/35

Operating systems provide a *file system*

- as an abstraction over physical storage devices (e.g. disks)
- providing named access to chunks of related data (files)
- providing access (sequential/random) to the contents of files
- allowing files to be arranged in a hierarchy of directories
- providing control over access to files and directories
- managing other meta-data associated with files (size, location, ...)

Operating systems also manage other resources

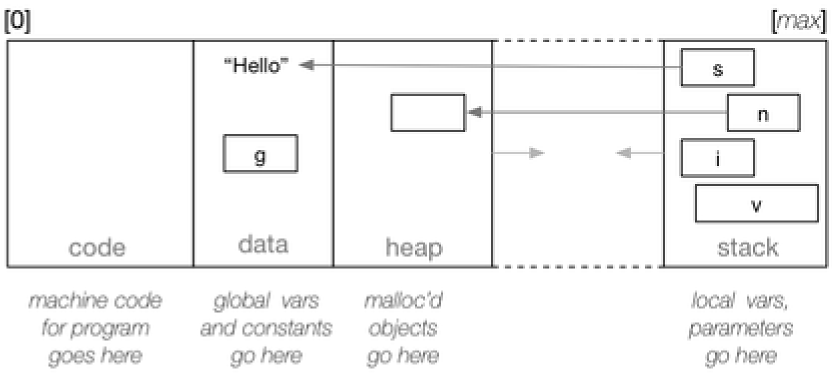
- memory, processes, processor time, i/o devices, networking, ...

Memory Management

Memory Management

17/35

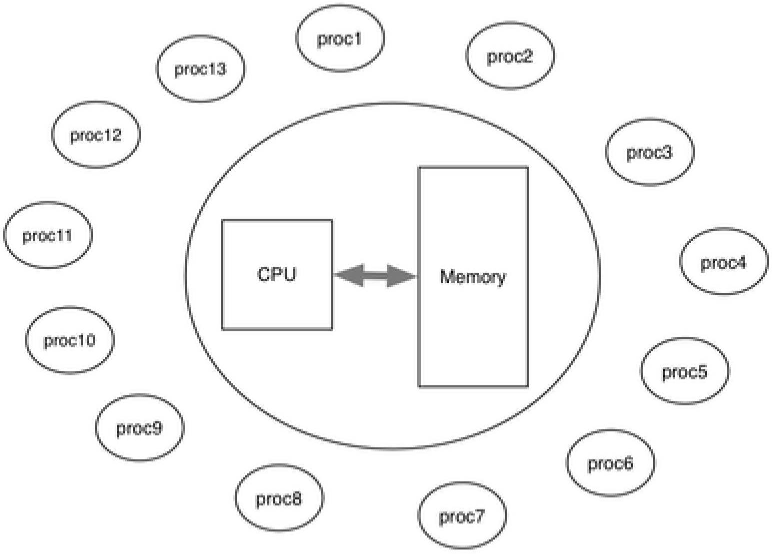
Operating system provides a view of memory for individual processes



... Memory Management

18/35

On a system with e.g. 1 CPU, 1 memory and 100's of processes



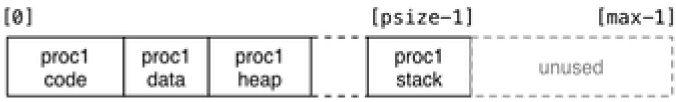
... Memory Management

19/35

The good-old-days ... one process/computation at a time

- the process can use the entire memory
- addresses within process code are absolute

Or, if the process did not need the entire memory



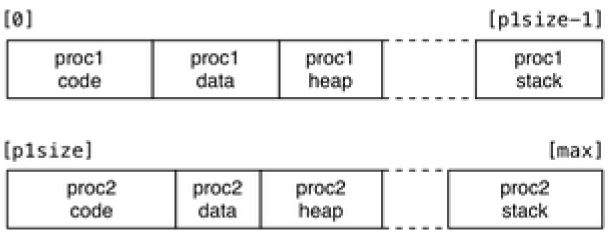
Easy to implement by initialising `$sp` to `psize-4`

... Memory Management

20/35

Two processes loaded into memory at once

- addreses in proc1 are absolute
- **all** addresses in proc2 need to be interpreted relative to `p1size`



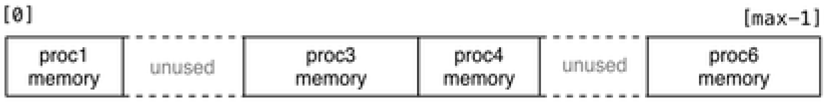
How to sort out proc2 addresses?

- replace them all by *addr*+`p1size` when loading process code+data
- each memory reference is mapped by (extra) hardware on-the-fly

... Memory Management

21/35

Consider a scenario where multiple processes are loaded in memory:



If we do on-the-fly address mapping, we need to ...

- remember base address for each process (process table)
- when process (re)starts, load base address into mapping hardware
- interpret every address *addr* in program as *base+addr*

Each process sees its own address space as [0 .. psize-1]

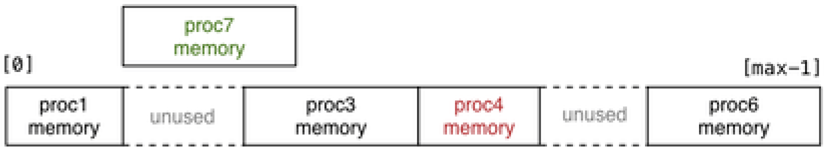
- so the process can be loaded anywhere in memory without change

---

... Memory Management

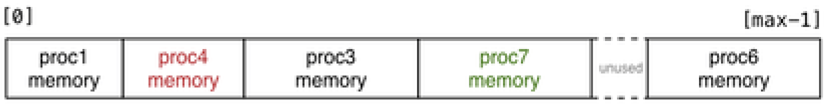
22/35

Consider the same scenario, but now we want to add a new process



The new process doesn't fit in any of the unused slots

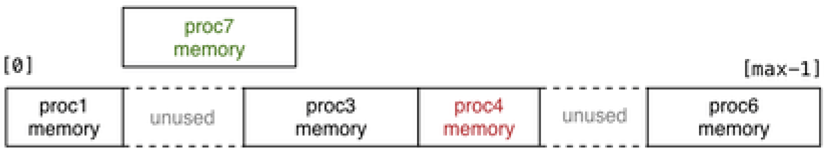
Could move some process to make a single large slot



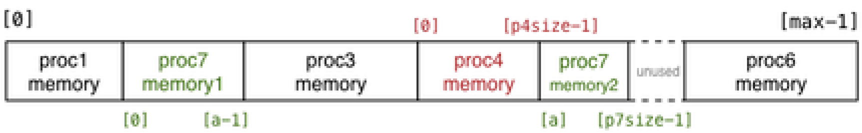
... Memory Management

23/35

Alternative strategy: split new process memory over two regions



becomes



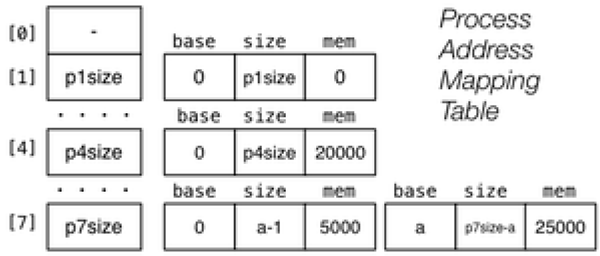
... Memory Management

24/35

Implications for splitting process memory across physical memory

- each chunk of process address space has its own *base*
- each chunk of process address space has its own *size*
- each chunk of process address space has its own *mem*ory location

Need a table of process/address information to manage this, e.g.



... Memory Management

25/35

Under this scheme, address mapping calculation is complicated

```
Address processToPhysical(pid, addr)
{
   Chunk chunks[] = getChunkInfo(pid);
   for (int i = 0; i < nChunks(pid); i++) {
      Chunk *c = &chunks[i];
      if (addr >= c->base && addr < c->base+c->size)
         break;
   }
   uint offset = addr - c->base;
   return c->mem + offset;
}
```

The above mapping *must* be done in hardware to be efficient.

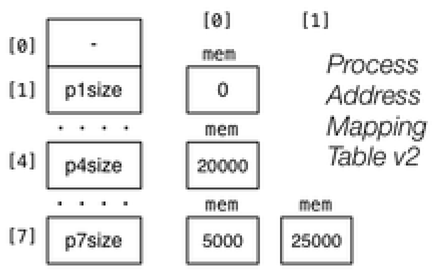
... Memory Management

26/35

Address mapping would be much simpler if all chunks were same size

- call each chunk of address space a *page*
- all pages are the same size *PageSize*
- process memory is spread across ⌈*ProcSize*/*PageSize*⌉ pages
- page *i* contains addresses in range *i*PageSize* .. *(i+1)*PageSize-1*

Also leads to a simpler address mapping table:



... Memory Management

27/35

And mapping from process address to physical address is very simple:

```
Address processToPhysical(pid, addr)
{
   PageInfo pages[] = getPageInfo(pid);
   uint pageno = addr / PageSize;  // int div
   uint offset = addr % PageSize;
   return pages[pageno].mem + offset;
}
```

Recall how integer division implemented in MIPS

- computation of `pageno` and `offset` is a single instruction

Note also that we are allowing for more complex `PageInfo` entries

Exercise 3: Address Mapping

28/35

Consider the scenario

- pages are 1KB in size
- a process requires (only) 4KB of memory
- pages are loaded: 0 @ 0x2000, 1 @ 0x4000, 2 @ 0x5000, 3 @ 0x0000

Show how the following process addreses are mapped to physical memory

- 0x0050
- 0x0260

Virtual Memory

29/35

A side-effect of this type of process→physical address mapping

- don't need to load all of processes pages up-front
- start with a small memory "footprint" (e.g. `main` + stack top)
- load new process address pages into memory *as needed*
- grow up to the size of the (available) physical memory

The strategy of ...

- dividing process memory space into fixed-size pages
- on-demand loading of process pages into physical memory

is called *virtual memory*

In this context, we call process addresses as *virtual addresses*.

... Virtual Memory

30/35

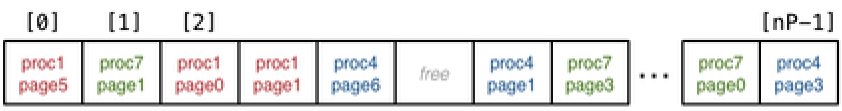
Page-sized regions of memory are called *page frames*

Page frames are typically 512B .. 8KB in size

In a 4GB memory, would have ≅4 million × 1KB page frames

Each page frame contains, a small region of a process's address space

Leads to a memory layout like this (with *nP* total pages of physical memory):



When a process completes, all of its page frames are released for re-use

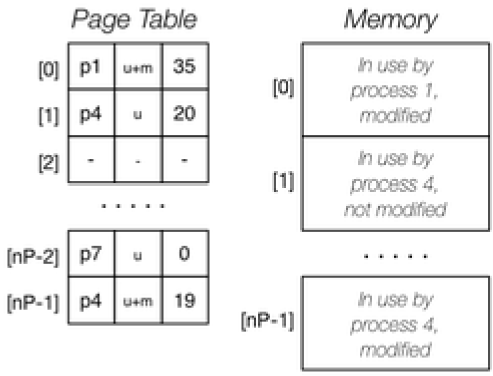
---

... Virtual Memory

31/35

Memory usage could be managed via a table, which records ...

- which process the page is allocated to (null if not in use)
- whether the page is currently in use and modified
- which chunk it represents within the process's address space



... Virtual Memory

32/35

Problem with above table

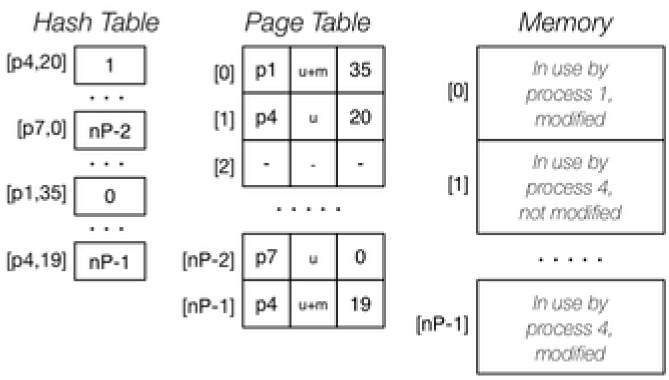
- page accesses give page number
- need to search table to find entry for page

```
typedef struct { int pid, char status, int pageno } PageData;

PageData PageTable[nP];  // one entry for each physical page

Address processToPhysical(pid, addr)
{
   int pageno = addr / PageSize;
   int offset = addr % PageSize;
   for (int i = 0; i < nP; i++) {
      PageData *p = PageTable[i];
      if (p->pid == pid && p->pageno == pageno)
         break;
   }
   return i*PageSize + offset;  // assumes page is loaded
}
```

... Virtual Memory

33/35

Search could be alleviated by *hashing*



... Virtual Memory

34/35

Which now gives an address mapping ...

```
typedef struct { int pid, char status, int pageno } PageData;

PageData PageTable[nP];  // one entry for each physical page

int HashTable[>nP];  // at least as many entries as PageTable

Address processToPhysical(pid, addr)
```

```
{
   int pageno = addr / PageSize;
   int offset = addr % PageSize;
   int key = hash(pid,pageno);  // index into HashTable
   int i = HashTable[key];      // index into PageTable
   PageData *p = PageTable[i];
   if (p->pid == pid && p->pageno == pageno)
      return i*PageSize + offset;
   else
      // hmmm ... this is not the page we want
}
```

For details on hashing, see COMP2521

... Virtual Memory

35/35

Alternatively, we can consider a per-process page table, e.g.

- each entry contains page status and physical address (if loaded)
- potentially, we need $\lceil ProcSize/PageSize \rceil$ entries in this table

```
typedef struct { char status, int memPage } PageData;

PageData *PageTables[maxProc];  // one entry for each process

Address processToPhysical(pid, addr)
{
   PageData *ProcPageTable = PageTables[pid];
   int pageno = addr / PageSize;
   int offset = addr % PageSize;
   PageData *p = ProcPageTable[pageno];
   if (loaded(p->status))
      return memPage*PageSize + offset;
   else
      // hmmm ... page not currently in memory
}
```

Produced: 17 Sep 2017