Week 11

Assignment 2

1/36

Page-based addressing ...

- process has a virtual address space from 0 .. StackTop (e.g. 10GB)
- physical memory has an address spce from 0 .. MemorySize (e.g. 4GB)
- virtual and physical memory manipulated via fixed-size chunks (e.g. 4KB)
- *pages* in virtual space map to *frames* in physical memory

Page fault ...

- referencing an address whose page is not currently held in memory
- allocate page to frame, using free frame (if any exists)
- otherwise evict some page (via *replacement policy*) and use its frame

... Assignment 2

2/36

Handling page faults must be done *fast*

⇒ Finding a victim page must be done *fast*

For LRU replacement, the following approach is ***not*** feasible

```
leastRecent = 0
oldestAccess = PageTable[0].accessed
foreach page P in PageTable {
   accessTime = PageTable[P].accessed
   if (accessTime < oldestAccess) {
      leastRecent = P
      oldestAccess = accessTime
   }
}
```

Time taken is proportional to size of PageTable (i.e. *O(n)*)

... Assignment 2

3/36

Solution: maintain an ordered list ...

- first entry in the list is always least recently used
- whenever a page is used, push to back of list

If list is structured properly, cost of each is constant (i.e. *O(1)*)

- needs just a few pointer rearrangements, not a scan

Note: currently unused pages are not in the list

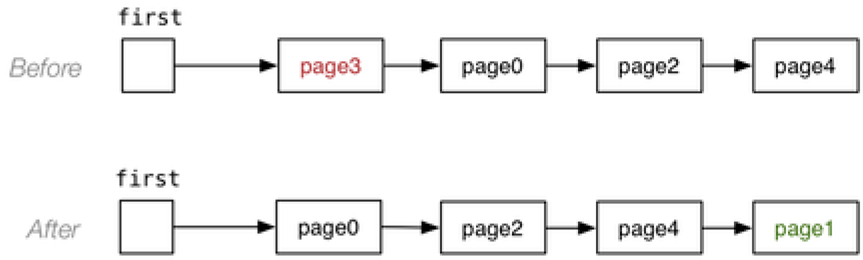
... Assignment 2

4/36

Finding a victim with an ordered list

- take the item off the front of the list
- make the (formerly) 2nd item into the new 1st item

Consider a request on page1, not currently loaded:



If we maintain a pointer to the `last` item, as well as `first`, no scan required

---

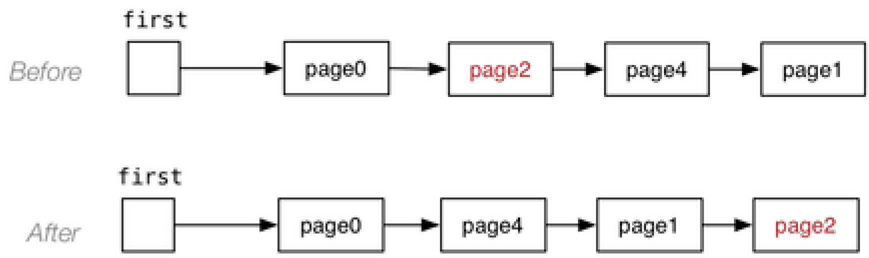
... Assignment 2

5/36

Updating list based on page usage

- remove the item from the middle of the list
- make the (formerly) 2nd item into the new 1st item

Consider a request on page2:



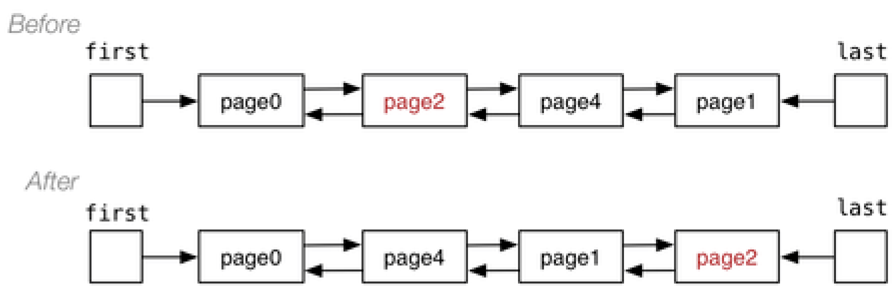
If we maintain a pointer to both `next` item and `prev` item, no scan required

---

... Assignment 2

6/36

Suitable list structure for both LRU and FIFO

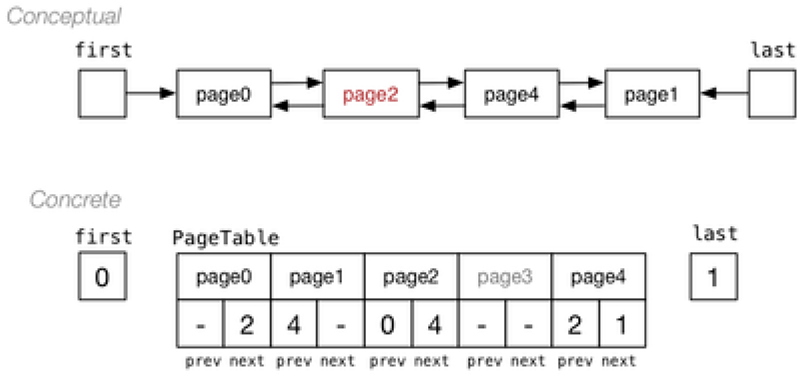


Note that "pointers" can actually be PageTable index values.

... Assignment 2

7/36

Concrete data structure for LRU list



I/O Devices

I/O Devices

9/36

Input/Output (I/O) devices

- allow programs to communicate with "the outside" world
- have significantly different characteristics to memory-based data

Memory-based data

- fast (ns) random access via (virtual) address
- transfer data in units of bytes, halfwords, words

Device data

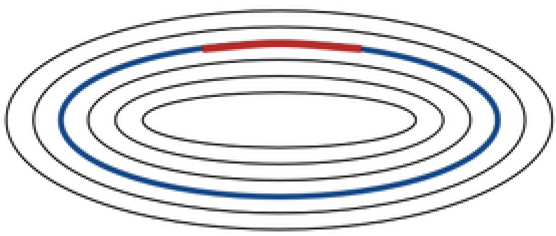
- much slower (ms) access, random or sequential
- transfer data in *blocks* (e.g. 512B, 4KB, ...)

... I/O Devices

10/36

Example: hard disk

- address specified by track and sector
- access time (move to track + wait for sector + read block)



Typical cost: 10ms seek + 5ms latency + 0.1ms transfer

... I/O Devices

11/36

Example: network transfer

- destination specified by IP address, packet size < 1KB
- transfer time includes
    - d_T transmission delay ... time push data packet onto "the wire"
    - d_P propagation delay ... time for packet to travel along "the wire"
    - d_C processing delay ... time to check header, re-route to next node
    - d_Q queueing delay ... time waiting on node before transmission
    - need to calculate for each of N hops, so $N(d_T + d_P + d_C + d_Q)$

Typical transfer time: 0.5ms (local ethernet), 200ms (internet), ...

Check transmission times using the `ping` command

... I/O Devices

12/36

Other types of devices ...

- keyboard ... byte-by-byte input, often line-buffered
- screen ... pixel-array output, typically via GPU
- mouse ... transmit X,Y movement and button presses
- camera ... convert video signal, frame-by-frame, to digital stream
- microphone ... convert analog audio signal to digital stream

Device Drivers

13/36

Each type of device has its own unique access protocol

- special control and data registers
- locations (buffers) for data to be read/written

Device drivers = code chunks to control an i/o device

- often written in assembler
- are core components of the operating system

Typical protocol to manipulate devices

- send request for operation (e.g. read, write, get status)
- receive interrupt when request is completed

For more details: see COMP2121 or ELEC2142

Memory-mapped I/O

14/36

Memory-mapped input/output

- operating system defines special memory address
- user programs perform i/o by getting/putting data into memory
- virtual memory addresses are associated with
  - data buffers of i/o device
  - control registers of i/o device

Advantages:

- uses existing memory access logic cirucits ⇒ less hardware
- can use full range of CPU operations on device memory

  (cf. having limited set of special instructions to manipulate i/o devices)

Devices on Unix/Linux

15/36

Unix treats devices uniformly as byte-streams (like files).

Devices can be accessed via the file system under `/dev`, e.g.

- **`/dev/diskN`** ... (part of) a hard drive

- **/dev/tty*N*** ... a terminal device
- **/dev/pty*N*** ... a pseudo-terminal device

Other interesting "devices" in `/dev`

- **/dev/mem** ... the physical memory (mostly protected)
- **/dev/null** ... data sink or empty source
- **/dev/random** ... stream of pseudo-random numbers

Exercise 1: Contents of pseudo-devices

16/36

How can we examine the contents of "devices" like ... ?

- **/dev/mem** ... the physical memory (mostly protected)
- **/dev/null** ... data sink or empty source
- **/dev/random** ... stream of pseudo-random numbers

... Devices on Unix/Linux

17/36

Two standard types of "device files" ...

Character devices (aka character special files)

- provide *unbuffered* direct access to hardware devices
- programmers interact with device by writing individual bytes
- do not necessarily provide byte-by-byte hardware i/o (e.g. disks)

Block devices (aka block special files)

- provide *buffered* access to hardware devices
- programmers interact with device by writing chunks of bytes
- data transferred to device via operating system buffers

... Devices on Unix/Linux

18/36

int ioctl(int *FileDesc*, int *Request*, void **Arg*)

- manipulates parameters of special files (behind open *`FileDesc`*)
- *`Request`* is a device-specific request code,
- *`Arg`* is either an integer modifier or pointer to data block
- requires `#include <sys/ioctl.h>`, returns 0 if ok, -1 if error

Example: SCSI disk driver

- `HDIO_GETGEO` ... get disk info in (heads,sectors,cylinders,...)
- `BLKGETSIZE` ... get device size in sectors
- in both cases, *`Arg`* is a pointer to an appropriate object

... Devices on Unix/Linux

19/36

int open(char **PathName*, int *Flags*)

- attempts to open file/device *`PathName`* in mode *`Flags`*
- *`Flags`* can specify caching, async i/o, close on `exec()`, etc.
- returns file descriptor if ok, -1 (plus `errno`) if error

ssize_t read(int *FileDesc*, void **Buf*, size_t *Nbytes*)

- attempts to read *`Nbytes`* of data into *`Buf`* from *`FileDesc`*
- returns # bytes actually read, 0 at EOF, -1 (plus `errno`) if error

ssize_t write(int *FileDesc*, void **Buf*, size_t *Nbytes*)

- attempts to write *`Nbytes`* of data from *`Buf`* to *`FileDesc`*
- returns # bytes actually written, -1 (plus `errno`) if error

Exercise 2: I/O Comparison

20/36

Compare char-by-char file i/o done as follows

```
while ((ch = getchar()) != EOF)
   putchar(ch);
```

vs

```
while (read(0, &ch, 1) != 0)
   write(1, &ch, 1);
```

Buffered I/O

21/36

Using a `read()` from a device for each byte is inefficient

- operating system uses a collection of buffers to hold data from device
- fed to user programs from buffer without accessing device each time

The standard i/o library also provides buffering of input

- from tty-like devices, generally line buffered
- from disk-like devices (files), read in `BUFSIZ` chunks
- buffering hidden from user, who sees `getchar()`, `fgets()`, etc.

Example `getc(fp)` implementation:

```
if (pos == BUFSIZ) {
   read(fileno(fp), buffer, BUFSIZ);
   pos = 0;
}
return buffer[pos++]
```

Exercise 3: Buffered I/O Implementation

22/36

Build a buffered i/o library like stdio

- `FILE *fopen(char *name, char mode)` ... mode = 'r' or 'w'
- `int getc(FILE *fp)` ... return value is char or EOF
- `putc(FILE *fp, char ch)` ... append ch to fp's stream
- `int fgets(FILE *fp, char *buf, int max)`

Assume a data structure like

```
typedef struct _file {
   char  mode;
   int   pos;
   char *buffer;
} FILE;
```

Exceptions

Exceptions

24/36

Exceptions are

- "unexpected" conditions occuring during program execution
- which require some form of action (maybe just quit)

Two types of exceptions

- exceptions ... from faults within an executing program
  - e.g. divide by zero, accessing invalid memory address, ...
  - often fatal to continued execution of program
- interrupts ... from events external to the program
  - e.g. i/o completion, signal from another process, ...
  - often, require some action and then execution can continue

---

... Exceptions

25/36

Many exceptions manifest themselves via signals

Signals are handled in a variety of ways

- Term ... terminate the process
- Ign ... ignore the signal
- Core ... terminate the process, dump core
- Stop ... stop the process
- Cont ... continue the process if currently stopped

Or you can write your own *signal handler*

See `man 7 signal` for details of signals and default handling.

... Exceptions

26/36

Signals from internal process activity, e.g.

- `SIGILL` ... illegal instruction
- `SIGABRT` ... generated by `abort()`
- `SIGFPE` ... floating point exception
- `SIGSEGV` ... invalid memory reference

Default handling of above signals: dump core and terminate process

Signals from external process events, e.g.

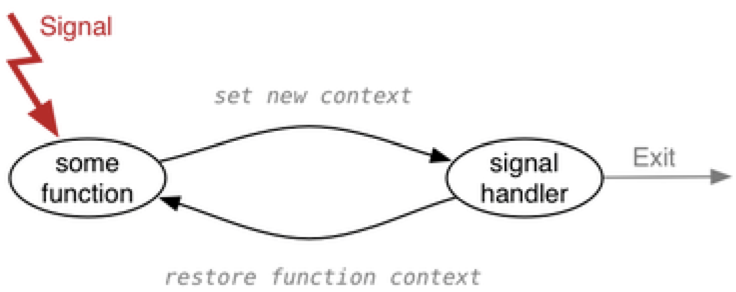
- `SIGINT` ... interrupt from keyboard (handled by Terminate)
- `SIGPIPE` ... broken pipe (handled by Terminate)
- `SIGCHLD` ... child process stopped or died (handled by Ingore)
- `SIGTSTP` ... stop typed at tty (control-Z) (handled by Stop)

Signal Handlers

27/36

Signal Handler = a function invoked in response to a signal

- should know which signal it was invoked by
- needs to ensure that invoking signal (at least) is blocked
- carries out appropriate action; may return



... Signal Handlers

28/36

```
int sigaction(int Signal, SigActStruct Action, ...)
```

- associates a handler with a signal, or sets `SIG_DFL` or `SIG_IGN`
- *`SigActStruct`* = **`struct sigaction`**, and contains
  - `sa_handler` ... pointer to signal handling function
  - `sa_sigaction` ... pointer to alternative handling function
  - `sa_mask` ... set of signals to be blocked in handler
  - `sa_flags` ... modifiers (e.g. don't block invoking signal)

`void (*sa_handler)(int)`

- takes a single argument (the invoking signal)

`void (*sa_sigaction)(int, siginfo_t *, void *)`

- first arg is invoking signal, others are context info (e.g. uid)

Exercise 4: Catching Signals

29/36

Write signal handlers that

1. catch signals HUP and TERM and print a message
2. catch SEGV when an invalid memory reference occurs
3. catch TERM, but use `sa_sigaction` to show uid

Note: you should *not* use `printf()` in signal handlers

Exceptions

30/36

Above exceptions are low-level, system ones.

Alternative notion of *exceptions*:

- unexpected conditions which arise during computation
- unexpected but *not* unanticipated (robust code)

Examples:

```
if ((p = malloc(sizeof(Type))) == NULL)
   ...

if (scanf("%d", &n) != 1)
   ...

avg = (n != 0) sum/n : 0;
```

... Exceptions

31/36

Such exceptions require handling in context of computation

Example:

- `create()` a data structure using multiple `malloc()`s
- part-way through function, one `malloc()` fails
- if abandoning `create()`, need to clean up previous `malloc()`s
- if terminating the entire process, no need to clean up

Many programming languages have special mechanisms for this

```
try { SomeCode } catch { HandleFailuresInCode }
```

C does not have generic exception handling; roll your own.

Interacting Processes

Interacting Processes

33/36

Processes can interact in several ways

- accessing the same resource (e.g. writing onto the same file)
- sending signals (via `kill()`)
- pipes: stdout of process A goes into stdin of process B
- sockets: using message passing (a la networks)

Uncontrolled interaction is a problem: non-deterministic

E.g. two processes writing to same file can have unpredictable results

- depends on actions of (opaque) scheduler

Mechanisms (e.g. `flock()`) exist to control interaction

File Locking

34/36

`flock(int FileDesc, int Operation)`

- controls access to shared files (note: files not fds)
- possible operations
  - **`LOCK_SH`** ... acquire shared lock
  - **`LOCK_EX`** ... acquire exclusive lock
  - **`LOCK_UN`** ... unlock
  - **`LOCK_NB`** ... operation fails rather than blocking
- `flock()` only *attempts* to acquire a lock
  - if it can't acquire the lock now, it is blocked (suspended)
  - when it can acuqire the lock, `flock()` returns
- only works correctly if all processes accessing file use locks
- return value: 0 in success, -1 on failure

... File Locking

35/36

If a process tries to acquire a *shared lock* ...

- if file not locked or other shared locks, OK
- if file has exclusive lock, blocked

If a process tries to acquire an *exclusive lock* ...

- if file is not locked, OK
- if any locks (shared or exclusive) on file, blocked

If using a non-blocking lock

- `flock()` returns 0 if lock was acquired
- `flock()` returns -1 if process would have been blocked

Exercise 5: Controlling access via flock()

36/36

Consider a program that

- forks a child
- both parent and child write to stdout

Without any controls, output is arbitrarily interleaved.

Using `flock()` control the file access

- so that each process writes a full line of output before the other

---

Produced: 12 Oct 2017