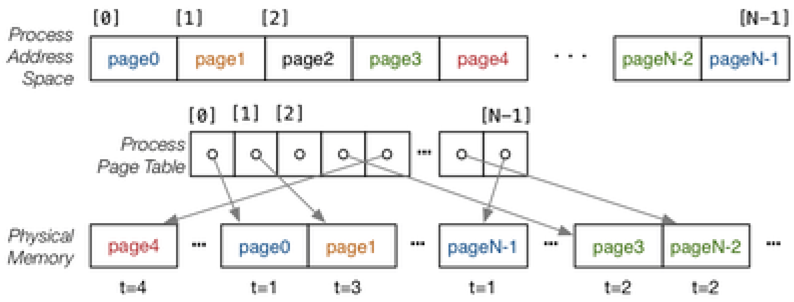
Week 09

Virtual Memory

1/41

Basic idea:

- process views its (virtual) *address space* as [0 .. maxAddr]
- memory manager partitions it over fixed-size *pages*
- process pages are loaded into memory when referenced
- process *page table* gives mapping virtual→physical pages



... Virtual Memory

2/41

Virtual address to memory address mapping:

```
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;
   int frameno;  // which page in memory
   if (loaded(ProcPageTable[pageno].status))
      frameno = ProcPageTable[pageno].memPage;
   else
      // load page into a free frame → frameno ...
   return frameno * PageSize + offset;
}
```

An Aside: Working Sets

3/41

Consider a new process commencing execution ...

- initially has zero pages loaded
- load page containing code for `main()`
- load page for `main()`'s stack frame
- load other pages when process references address within page

Do we ever need to load all process pages at once?

... An Aside: Working Sets

4/41

From observations of running programs ...

- in any given window of time,
  a process is likely to access only a small subset of its pages

Known as the *working set* model (cf *locality of reference*)

Only need to hold, at any given time, the process's working set of pages

Implications:

- if each process has a relatively small working set,
  can hold pages for many active processes in memory at same time
- if only need to hold some of process's pages in memory,
  process address space can be larger than physical memory

Exercise 1: Reference Locality

5/41

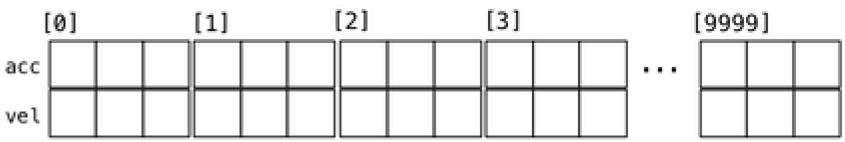
Consider the following data structure

```
typedef struct { int acc[3]; int vel[3]; } Point
Point p[10000];
```

and two functions to clear all the values in the array.

Discuss the locality of data reference in each of these functions:

```
void clear1(point *p, int n)
{
   int i,j;
   for (i = 0; i < n; i++) {
      for (j = 0; j < 3; j++) p[i].acc[j] = 0;
      for (j = 0; j < 3; j++) p[i].vel[j] = 0;
   }
}
void clear2(point *p, int n)
{
   int i,j;
   for (j = 0; i < 3; j++) {
      for (i = 0; i < n; i++) p[i].acc[j] = 0;
      for (i = 0; i < n; i++) p[i].vel[j] = 0;
   }
}
```



Virtual Memory

6/41

We say that we "load" pages into physical memory

But where are they loaded from?

- code is loaded from the executable file stored on disk
- global data is also initially loaded from here
- dynamic (heap, stack) data is created in memory

Consider a process whose address space exceeds physical memory

The pages of dynamic data not currently in use

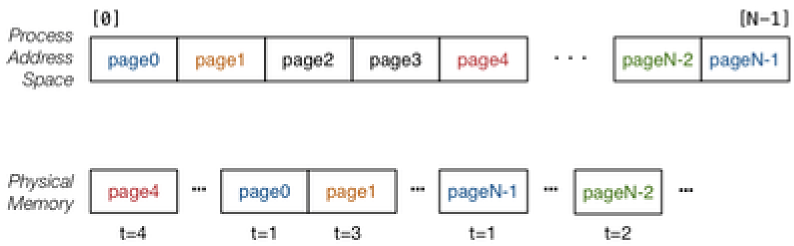
- may need to be removed temporarily from memory (see later)
- thus would also be saved on disk and restored from disk

... Virtual Memory

7/41

We can imagine that a process's address space ...

- exists on disk for the duration of the process's execution
- and only some parts of it are in memory at any given time



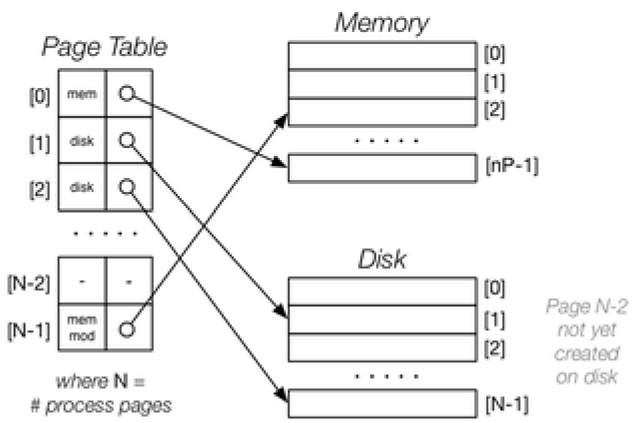
Transferring pages between disk↔memory is **very** expensive

- need to ensure minimal reading from / writing to disk

... Virtual Memory

8/41

Per-process page table, allowing for some pages to be not loaded



... Virtual Memory

9/41

Recall the address mapping process with per-process page tables

```
Address processToPhysical(pid, addr)
{
   PageData *ProcPageTable = PageTables[pid];
   int pageno = addr / PageSize;
   int offset = addr % PageSize;
   int frameno;  // which page in memory
   if (loaded(ProcPageTable[pageno].status))
      frameno = ProcPageTable[pageno].memPage;
   else
      // load page into a free frame → frameno ...
   return frameno * PageSize + offset;
}
```

What to do if the page is not loaded?

Page Faults

10/41

The scenario of requesting a non-loaded page is a *page fault*.

One approach to handling a page fault ...

- find a free (unused) page frame in memory and use that

```
// load page into a free frame ...
else {
```

```
        frameno = getPageFrame();
        p->memPage = frameno;
        p->status  = LOADED;
    }
```

Assumes that we have a way of quickly identifying free page frames

Commonly handled via a *free list*

Reminder: pages allocated to a process become *free* when the process exits

---

... Page Faults

11/41

What happens if there are currently no free page frames

What does `getPageFrame()` do?

Possibilities:

- *suspend* the requesting process until a page is freed
- *replace* one of the currently loaded/used pages

Suspending requires the process manager to

- maintain a (priority) queue of processes waiting for pages
- dequeue and schedule the first process on queue when page freed

Will discuss process queues further in next section.

Page Replacement

12/41

What happens when a page is replaced?

- if it's been modified since loading, save to disk **
  (in the disk-based virtual memory space of the running process)
- grab its frame number and give it to the requestor

How to decide which frame should be replaced?

- maintain a "usefulness" measure for each frame
- grab the frame with lowest usefulness
  (could manage via a priority queue ... see COMP2521)

** we now need a flag to indicate whether a page is modified

```
#define LOADED 0x00000001
#define MODIFIED 0x00000002
```

... Page Replacement

13/41

Factors to consider in deciding which page to replace

- best page is one that won't be used again by its process
- prefer pages that are read-only (no need to write to disk)
- prefer pages that are unmodified (no need to write to disk)
- prefer pages that are used by only one process (see later)

OS can't predict whether a page will be required again by its process

But we do know whether it has been used recently (if we record this)

Useful heuristic: *LRU replacement*

- a page not used recently may not be needed again soon

The working set model helps virtual memory systems avoid *thrashing*

... Page Replacement

14/41

LRU is one replacement strategy. Others include:

First-in-first-out (FIFO)

- page frames are entered into a queue when loaded
- page replacement uses the frame from the front of the queue

Clock sweep

- maintains a referenced bit for each frame, updated when page is used
- maintains a circular list of allocated frames
- uses a "clock hand" which iterates over page frame list
- replacement uses first-found unreferenced frame
    - skipping over and resetting referenced bit in all referenced pages

Exercise 2: Page Replacement

15/41

Show how the page frames and page tables change when

- there are 4 page frames in memory
- the process has 6 pages in its virtual address space
- a LRU page replacement strategy is used

For each of the following sequences of virtual page accesses

1. 0, 5, 0, 0, 5, 1, 5, 1, 2, 4, 3, 3, 4, 2, 5, 3, 2, ...
2. 5, 0, 0, 0, 5, 1, 1, 5, 1, 5, 2, 2, 3, 0, 0, 5, ...

Assume that all page tables and page frames are empty/unused

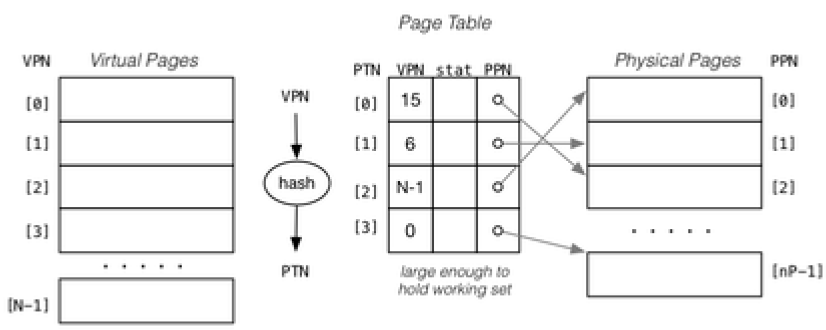
Virtual Memory

16/41

Page tables (PTs) revisited ...

- a virtual address space with N pages needs N PT entries
- since N may be large, do not want to store whole PT
- especially since working set tells us $n \ll N$ needed at once

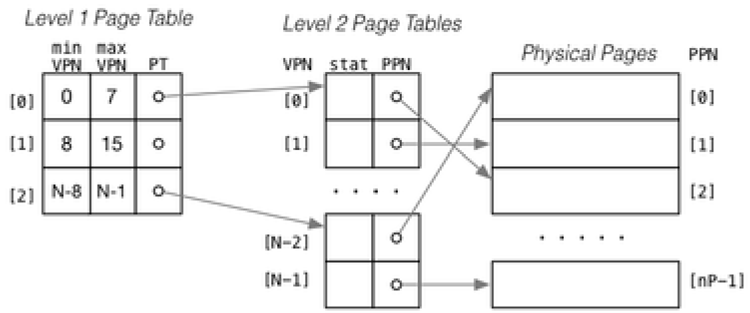
One possibility: PT with $n < N$ entries and hashing



... Virtual Memory

17/41

Alternative strategy: multi-level page tables



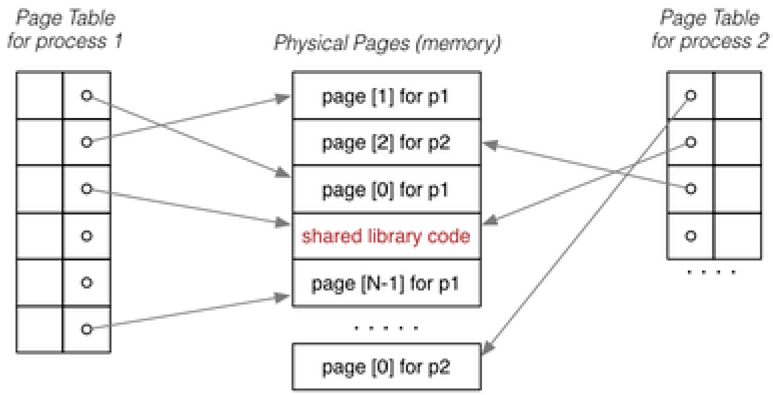
Effective because not all pages in virtual address space are required
(e.g. the pages between the top of the heap and the bottom of the stack)

... Virtual Memory

18/41

Virtual memory allows sharing of read-only pages (e.g. library code)

- several processes include same frame in their virtual address space



Assignment 2

19/41

Simulation of virtual memory management system

- representation for physical and virtual memory spaces
- read process traces (sequences of page references)
- implement effect of each reference on pages/frames
- collect and display statistics
- do this for different page replacement strategies

Process trace: sequence of *(mode,page#)*, where *mode* is "r" or "w"

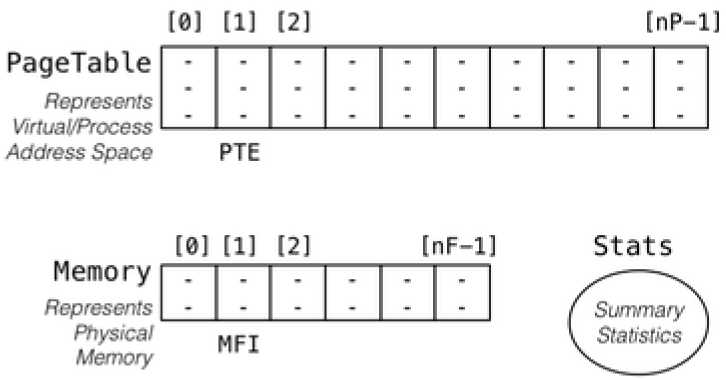
```
r0 w3 r1 r0 r2 w3 r0 w3 r2 w2 r0 w3 r0 w2 r0 r1
```

Each *(mode,page#)* represents an action over one clock tick

... Assignment 2

20/41

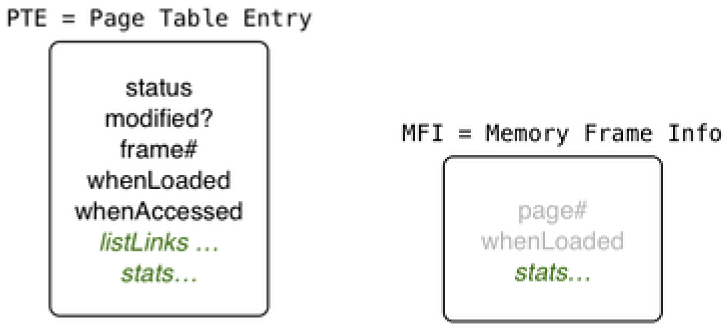
Data structures representing address spaces



... Assignment 2

21/41

Data structures representing pages and frames



... Assignment 2

22/41

Main program ...

```
collect (from argv) Policy, #Pages, #Frames
initialise Stats, PageTable, Memory
time = 0
while (getNextReference(&mode,&page#))
{
   // mode is either r or w
   // page# is value in 0..#Pages-1
   requestPage(page#, mode, time)
   time++
}
show summary stats
show frame stats
show page table stats
```

... Assignment 2

23/41

Page request handler ...

```
int requestPage(page#, mode, time)
{
   P = PageTable[page#]
   switch (status of P)
   case IN_MEMORY:
      pageHit
   case NOT_USED:
   case ON_DISK:
      pageFault
      F = find a frame for P to use
      set P's frame to F
   }
   if (mode is write) set P to modified
   update P's access time
   return P's frame#
}
```

... Assignment 2

24/41

Finding a frame F for a page P to use

```
if (there are free frames)
   F = one of the free frames
else {
   V = find a page to replace using Policy
   if (V is modified) save to disk
   F = V's frame
   load P's data into F (from disk)
   reset P's data (modified, etc.)
}
```

... Assignment 2

25/41

Finding a page to replace using LRU ...

```
oldest = now
victim = NONE
for (i = 0; i < #Pages; i++) {
   P = PageTable[i]
   if (P's accessTime < oldest)
      oldest = P's accessTime
      victim = P's page
   }
}
```

Much better if we have a list of pages ordered by load time

- when we (re)load P, move P to the back of the list

... Assignment 2

26/41

Finding a page to relace using FIFO ...

```
oldest = now
victim = NONE
for (i = 0; i < #Pages; i++) {
   P = PageTable[i]
   if (P's loadTime < oldest)
      oldest = P's loadTime
      victim = P's page
   }
}
```

Much better if we have a list of pages ordered by access time

- on access to P, move P to the back of the list

... Assignment 2

27/41

Your task ...

- add statistics collection at appropriate points
- convert linear scans to appropriate lists
- do this for LRU and FIFO, challenge: do Clock as well

Processes

Processes

29/41

A *process* is an instance of an executing program

- static information: program code and data
- dynamic state: heap, stack, registers, program counter
- OS-supplied state: environment variables, stdin, stdout

Process management forms a critical component of OS functionality

The OS provides processes with

- control-flow independence
  - the process executes as if it is the only process running on the machine
- private address space
  - the process has its own address space, possibly larger than physical memory

---

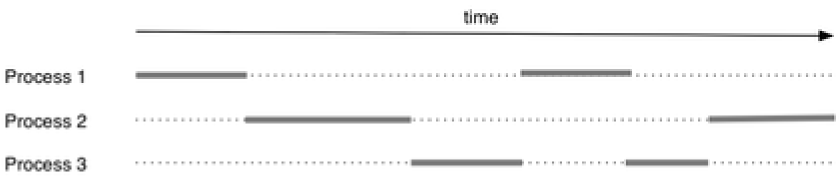
... Processes

30/41

Control-flow independence ("I am the only process, and I run until I finish")

In reality, there are multiple processes running on the machine

- each process uses the CPU until *pre-empted* or exits
- then another process uses the CPU until it too is pre-empted
- eventually, the first process will get another run on the CPU



Overall impression: three programs running simultaneously

... Processes

31/41

What can cause a process to be pre-empted?

- it runs "long enough" and the OS replaces it by a waiting process
- it attempts to perform a long-duration task, like i/o

On pre-emption ...

- the process's entire dynamic state must be saved (incl PC)
- the process is flagged as temporarily suspended
- it is placed on a process (priority) queue for re-start

On resuming, the state is restored and the process starts at saved PC

Overall impression: I ran until I finished all my computation

---

OS Process Management

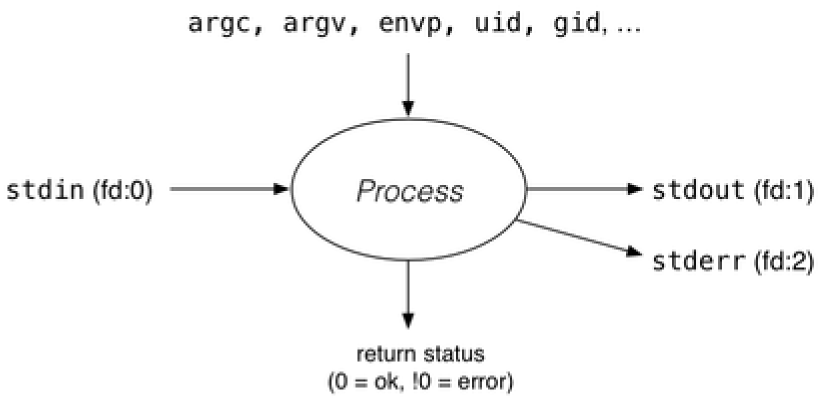
32/41

OSs maintain information about processes in *process control blocks*

Unix/Linux Processes

33/41

Context for processes running on Unix/Linux systems



... Unix/Linux Processes

34/41

Unix provides a range of tools for manipulating processes

Commands:

- **`sh`** ... for creating processes via object-file name
- **`ps`**, **`w`**, **`top`** ... show process information
- **`kill`** ... send a signal to a process

System calls:

- **`fork()`** ... create a new child process (copy of current process)
- **`execve()`** ... convert one process into another by executing object
- **`wait()`** ... wait for state change in child process
- **`kill()`** ... send a signal to a process
- **`_exit()`** ... terminate an executing process (after clean up)

Exercise 3: Process Information

35/41

How can I find out ...

- what processes I currently have running
- what are all of the processes running on the system
- what are the top CPU-using processes
- who's logged in and what they're doing

... Unix/Linux Processes

36/41

Information associated with processes:

- `pid` ...process id, unique among current processes
- `ruid`, `euid` ... real and effective user id
- `rgid`, `egid` ... real and effective group id
- current working directory
- accumulated execution time (user/kernel)
- user file descriptor table
- information on how to react to signals
- pointer to process page table
- process state ... running, suspended, asleep, etc.

This data is split across a kernel process table entry and a user area

---

Process-related System Calls

37/41

`pid_t fork(void)`

- creates new process by duplicating the calling process
- new process is the *child*, calling process is the *parent*
- child has a different process ID (pid) to the parent
- in the child, `fork()` returns 0
- in the parent, `fork()` returns the pid of the child
- if the system call fails, `fork()` returns -1
- child inherits copies of parent's address space and open fd's
- child does not inherit copies of pending signals, memory locks, ...

Typically, the child pid is a small increment over the parent pid

---

... Process-related System Calls

38/41

Every process in Unix/Linux is allocated a process ID

- a +ve integer, unique among currently executing processes
- with type **`pid_t`** (defined in <unistd.h>)
- process 0 is the *scheduler* (part of kernel)
- process 1 is `init` (for starting/stopping the system)

- low-numbered processes are typically system-related
- regular processes have PID in the range 300 .. MaxPid (e.g. 2^{16})

Processes are also collected into *process groups*

- each group is associated with a unique PGID
- groups allow distribution of signals to a set of related processes
- one important application: job control (control-Z)

... Process-related System Calls

39/41

`pid_t getpid()`

- returns the process ID of the current process

`pid_t getppid()`

- returns the parent process ID of the current process

`pid_t getpgid(pid_t pid)`

- returns the process group ID of specified process
- if `pid` is zero, use get PGID of current process

`int setpgid(pid_t pid, pid_t pgid)`

- set the process group ID of specified process

All return -1 and set `errno` on failure.

... Process-related System Calls

40/41

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

- pause current process until process `pid` changes state
    - where state changes include finishing, stopping, re-starting, ...
- ensures that child resources are released on exit
- special values for `pid` ...
    - if `pid` = -1, wait on any child process
    - if `pid` = 0, wait for any child in process group
    - if `pid` > 0, wait on the specified process
- information about child process state is stored in `status`
- `options` alters behaviour of `wait` (e.g. don't block)

`pid_t wait(int *status)`

- equivalent to `wait(-1, &status, 0)`

Exercise 4: Forking, etc.

41/41

Write a small program that

- `fork()`s a child process
- gets both processes to print details about themselves
- explores the use of `wait()` to control sequencing

```
int main(void)
{
   pid_t pid = fork();
   if (pid != 0)
      // parent actions
   else
      // child actions
   return 0;
}
```

Produced: 21 Sep 2017