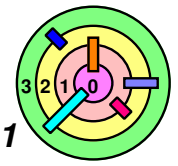
Warmup #1

Bill Cheng

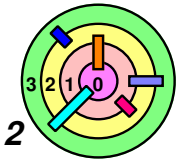
http://merlot.usc.edu/william/usc/



Discussion Sections

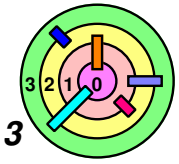
IMPORTANT:

- please understand that discussion section material are ***NOT*** substitute for reading the specs and the grading guidelines
  - you are expect to read the ***specs***
  - you are expect to read the ***requirements*** the specs refer to
  - you are expect to read the ***grading guidelines***
  - it's your responsibility



Programming & Good Habbits

- ***Always*** **check return code!**
  - `open(), write()`
  - `malloc()`
  - `switch (errno) { ... }`
- **Initialize *all* variables!**
  - `int i=0;`
  - `char *p=NULL;`
  - `struct timeval timeout;`
    `memset(&timeout, 0, sizeof(struct timeval));`
- ***Never*** **leak any resources!**
  - `malloc() and free()`
  - `open() and close()`
  - delete temporary files

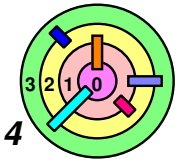


Programming & Good Habbits

- ***Don't*** **assume external input will be short**
  - use `strncpy()` and not `strcpy()`
  - use `snprintf()` and not `sprintf()`
  - use `sizeof()` and not a constant, for example,

```
unsigned char buf[80];

buf[0] = '\0'; /* initialization */
strncpy(buf, argv[1], sizeof(buf));
buf[sizeof(buf)-1] = '\0'; /* in case argv[1] is long */
```

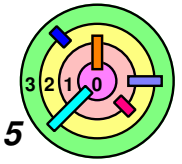
- **Fix your code so that you have *zero* compiler warnings!**
  - use `-Wall` when you compile to get all compiler warnings



Notes on gdb

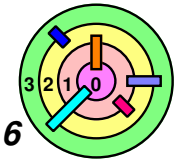
The debugger is your friend! Get to know it *NOW*!

compile program with:	`-g`
start debugging:	`gdb [-tui] listtest`
set breakpoint:	`(gdb) break main`
	`(gdb) break listtest.c:87`
run program (w/ arguments):	`(gdb) run [arg1 arg2 ...]`
clear breakpoint:	`(gdb) clear`
stack trace:	`(gdb) where`
print field:	`(gdb) print pList->anchor`
print in hex:	`(gdb) print/x pList->anchor`
single-step at same level:	`(gdb) next`
single-step into a function:	`(gdb) step`
print field after every cmd:	`(gdb) display pList->num_members`
assignment:	`(gdb) set pList->num_members=99`
continue:	`(gdb) cont`
quit:	`(gdb) quit`



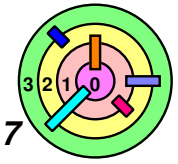
Warmup #1

- 2 parts
  - develop a *doubly-linked circular list* called *My402List*
    - this corresponds to part (A) of the grading guidelines
    - to implement a traditional *linked-list abstraction*
      - internally, the implementation is a *circular list*
      - internally, it behaves like a *traditional list*
      - why? circular list implementation may be a little "cleaner"
  - use your doubly-linked circular list to implement a command:
    - *sort* - sort a list of bank transactions
    - this corresponds to part (B) of the grading guidelines



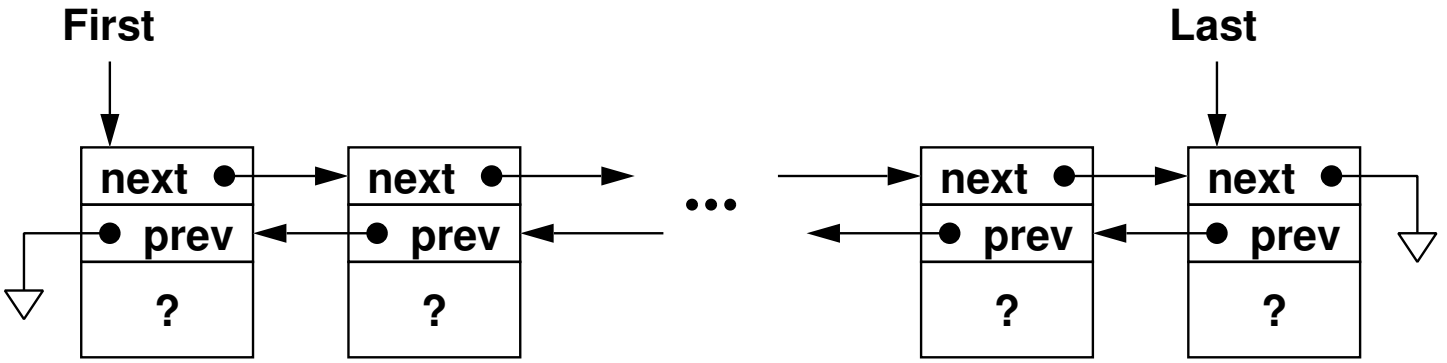
A Linked-List Abstraction

- A list of elements, linked so that you can move from one to the next (and/or previoius)
  - each element holds an object of some sort
- *Functionally:*
  - First()
  - Next()
  - Last()
  - Prev()
  - Insert()
  - Remove()
  - Count()
- Need to have a well-defined interface
  - once you have a good interface, if the implementation is broken, fix the implementation!
    - don't fix the "application"

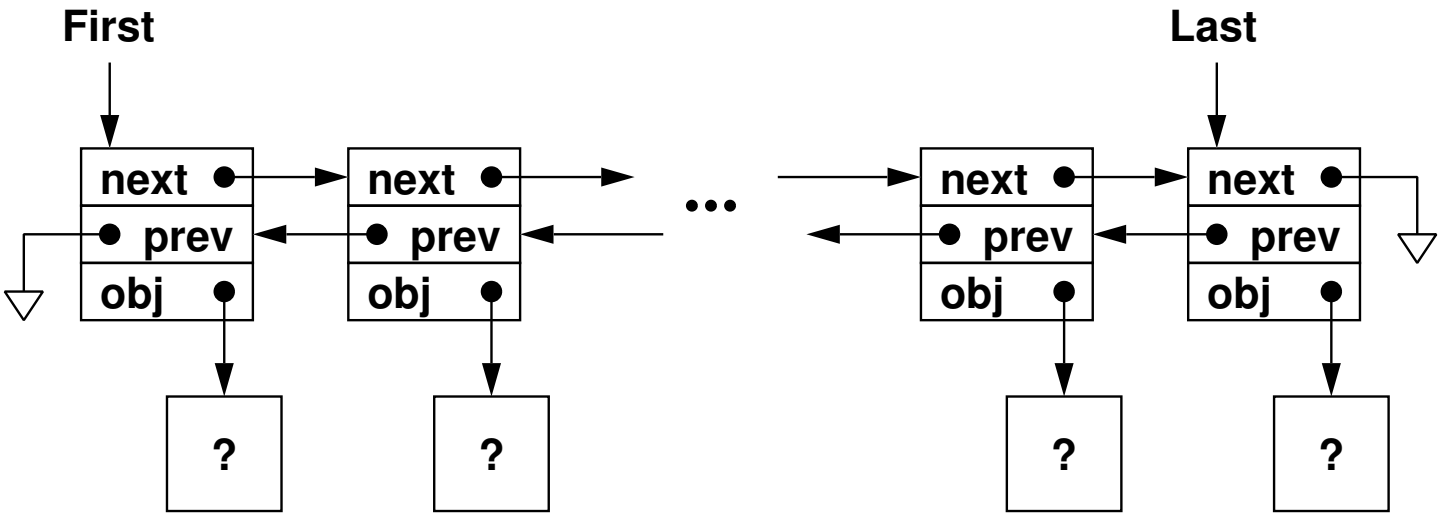


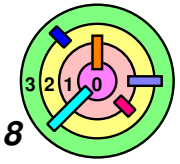
A Linked-List Abstraction

- There are basically two types of lists
  1) next/prev pointers in object
  2) next/prev pointers outside of object
- (1) has a major drawback that a list item cannot be inserted into multiple lists



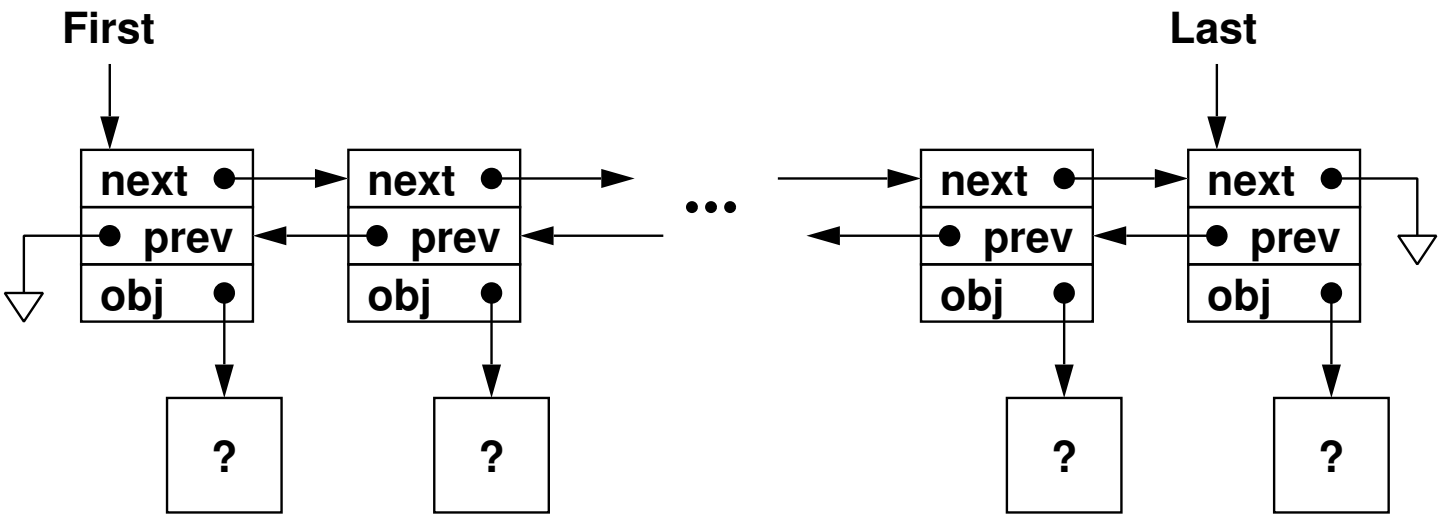
- we will implement (2) in warmup1, our kernel uses (1)





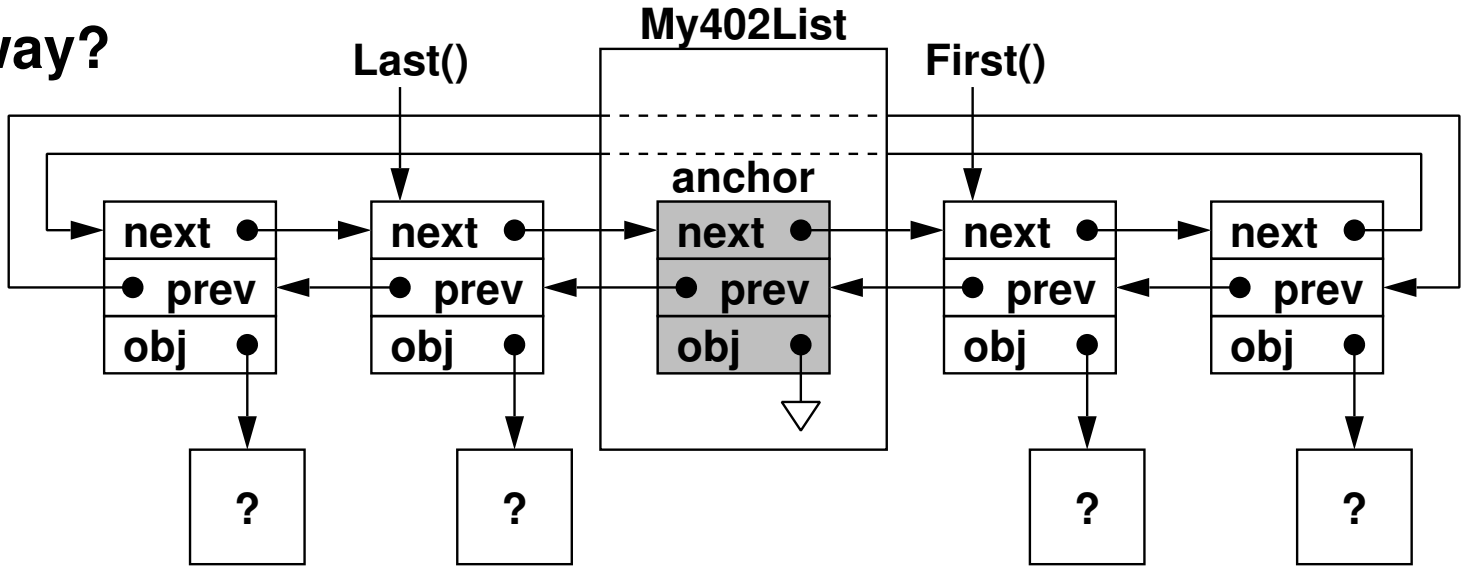
Doubly-linked Circular List

Abstraction

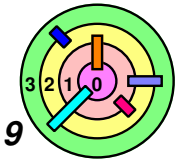


Implementation

- why this way?



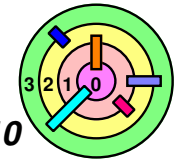
- your job is to implement the traditional list abstraction using a circular list



my402list.h

```
#ifndef _MY402LIST_H_
#define _MY402LIST_H_

#include "cs402.h"

typedef struct tagMy402ListElem {
    void *obj;
    struct tagMy402ListElem *next;
    struct tagMy402ListElem *prev;
} My402ListElem;

typedef struct tagMy402List {
    int num_members;
    My402ListElem anchor;

    /* You do not have to set these function pointers */
    int   (*Length)(struct tagMy402List *);
    int   (*Empty)(struct tagMy402List *);

    int   (*Append)(struct tagMy402List *, void*);
    int   (*Prepend)(struct tagMy402List *, void*);
    void  (*Unlink)(struct tagMy402List *, My402ListElem*);
    void  (*UnlinkAll)(struct tagMy402List *);
```

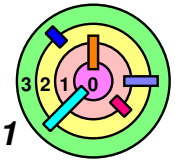
- You need to learn to ignore things you don't understand
  - assume that they are perfect



my402list.h

```
    int  (*InsertBefore)(struct tagMy402List *, void*, My402ListElem*);
    int  (*InsertAfter)(struct tagMy402List *, void*, My402ListElem*);

    My402ListElem *(*First)(struct tagMy402List *);
    My402ListElem *(*Last)(struct tagMy402List *);
    My402ListElem *(*Next)(struct tagMy402List *, My402ListElem *cur);
    My402ListElem *(*Prev)(struct tagMy402List *, My402ListElem *cur);

    My402ListElem *(*Find)(struct tagMy402List *, void *obj);
} My402List;

extern int  My402ListLength(My402List*);
extern int  My402ListEmpty(My402List*);

extern int  My402ListAppend(My402List*, void*);
extern int  My402ListPrepend(My402List*, void*);
extern void My402ListUnlink(My402List*, My402ListElem*);
extern void My402ListUnlinkAll(My402List*);
extern int  My402ListInsertAfter(My402List*, void*, My402ListElem*);
extern int  My402ListInsertBefore(My402List*, void*, My402ListElem*);

extern My402ListElem *My402ListFirst(My402List*);
extern My402ListElem *My402ListLast(My402List*);
extern My402ListElem *My402ListNext(My402List*, My402ListElem*);
extern My402ListElem *My402ListPrev(My402List*, My402ListElem*);

extern My402ListElem *My402ListFind(My402List*, void*);

extern int My402ListInit(My402List*);
#endif /*_MY402LIST_H_*/
```

You need to implement all the mentioned functions

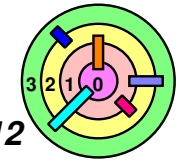


my402list.c

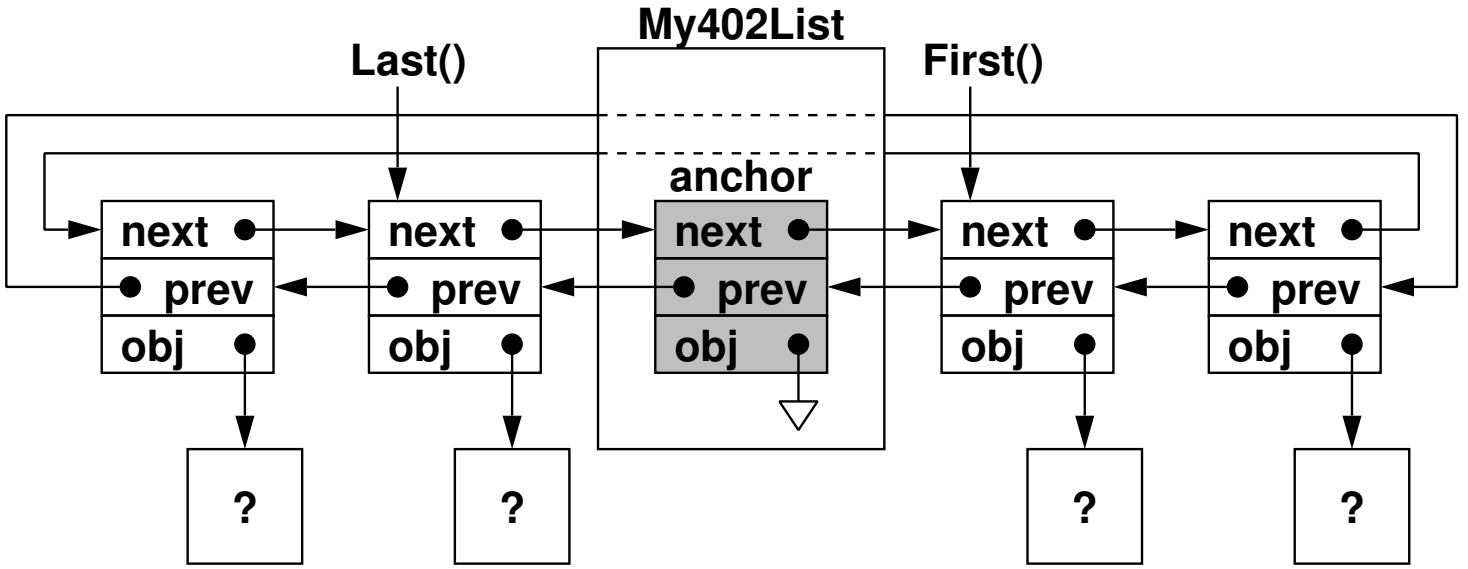
How to start?

- `cp my402list.h my402list.c`
- edit `my402list.c` in a text editor
  - replace data structure declarations with "`#include`"
  - change all function declarations to function implementations
    - remove "`extern`" and implement function

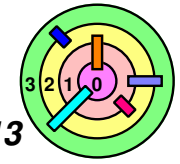
```
#include "my402list.h"

extern int  My402ListLength(My402List*);
extern int  My402ListEmpty(My402List*);

extern int  My402ListAppend(My402List*, void*);
extern int  My402ListPrepend(My402List*, void*);
extern void My402ListUnlink(My402List*, My402ListElem*);
extern void My402ListUnlinkAll(My402List*);
extern int  My402ListInsertAfter(My402List*, void*, My402ListElem*);
extern int  My402ListInsertBefore(My402List*, void*, My402ListElem*);

extern My402ListElem *My402ListFirst(My402List*);
extern My402ListElem *My402ListLast(My402List*);
extern My402ListElem *My402ListNext(My402List*, My402ListElem*);
extern My402ListElem *My402ListPrev(My402List*, My402ListElem*);

extern My402ListElem *My402ListFind(My402List*, void*);

extern int My402ListInit(My402List*);
```



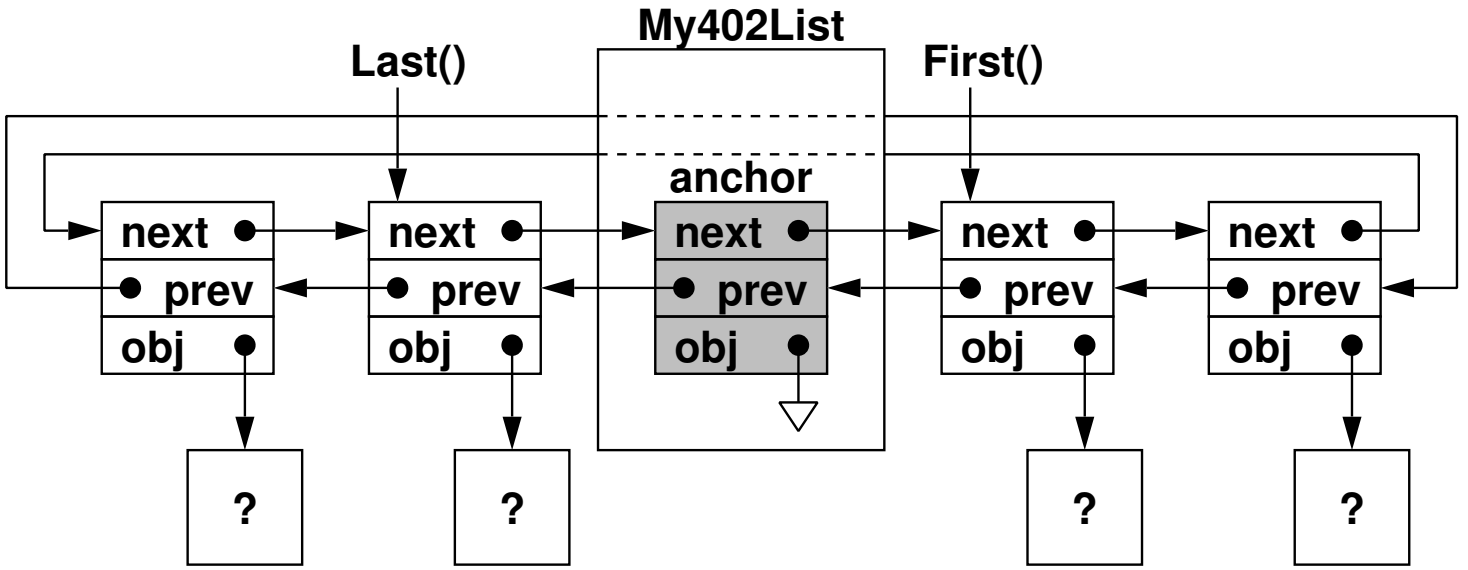
Implementation




```
int Length() { return num_members; }
int Empty() { return num_members<=0; }

int Append(void *obj);
int Prepend(void *obj);
void Unlink(My402ListElem*);
void UnlinkAll();
int InsertBefore(void *obj, My402ListElem *elem);
int InsertAfter(void *obj, My402ListElem *elem);

My402ListElem *First();
My402ListElem *Last();
My402ListElem *Next(My402ListElem *cur);
My402ListElem *Prev(My402ListElem *cur);

My402ListElem *Find(void *obj);
int Init();
```



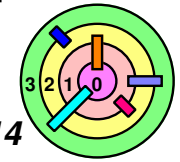
Usage - Traversing the List




```
void Traverse(My402List *list)
{
  My402ListElem *elem=NULL;

  for (elem=My402ListFirst(list);
       elem != NULL;
       elem=My402ListNext(list, elem)) {
    Foo *foo=(Foo*)(elem->obj);

    /* access foo here */
  }
}
```

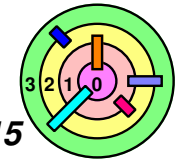
⇨ This is how an *application* will use `My402List`

- you must support this *"contract"* with you application
- if broken, fix the "implementation" and not the "application"



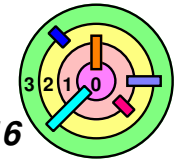
listtest

- Use provided `listtest.c` and `Makefile` to create `listtest`
  - `listtest` must run without error and you must not change `cs402.h`, `my402list.h`, `listtest.c` and `Makefile`
  - they specify how your code in `my402list.c` is expected to be used
- You should learn how to run `listtest` under `gdb`



Sort Command

- `warmup1 sort [tfile]`
  - produce a sorted transaction history for the transaction records in `tfile` (or `stdin`) and compute balances
- Input is an ASCII text file (use `fgets()` to read a line)
  - each line in a `tfile` contains 4 fields delimited by `<TAB>`
    - transcation type (single character)
      - `"+"` for deposit
      - `"-"` for withdrawal
    - transcation time (UNIX time)
      - `man -s 2 time`
    - amount (a number, a period, two digits)
    - transcation description (textual description)
      - cannot be empty
- Output must be in the specified format exactly
  - use the grading guidelines to check if you miss something
    - formatting bugs should be very easy to fix



Sort Command

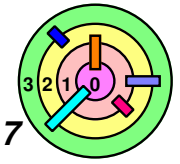
Output

```
0000000001111111111222222222233333333334444444444555555555566666666667777777778
1234567890123456789012345678901234567890123456789012345678901234567890123456789

+----------------+--------------------------+----------------+----------------+
|       Date     | Description              |         Amount |        Balance |
+----------------+--------------------------+----------------+----------------+
| Thu Aug 21 2008 | ...                     |       1,723.00 |       1,723.00 |
| Wed Dec 31 2008 | ...                     |  (       45.33) |       1,677.67 |
| Mon Jul 13 2009 | ...                     |      10,388.07 |      12,065.74 |
| Sun Jan 10 2010 | ...                     |  (      654.32) |      11,411.42 |
+----------------+--------------------------+----------------+----------------+
```

How to keep track of balance

- first thing that comes to mind is to use `double`
- the weird thing is that if you are not very careful with double, your output will be wrong (by 1 penny) once in a while
- recommendation: keep the balance in cents, not dollars
  - no precision problem with integers!

Read *grading guidelines* and find many examples of *valid input* and *expected printout*

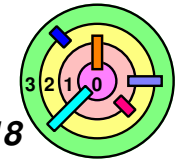


Sort Command

```
000000000111111111122222222223333333333444444444455555555556666666666777777777 8
123456789012345678901234567890123456789012345678901234567890123456789012345678 90

+----------------+--------------------------+----------------+----------------+
|       Date     | Description              |         Amount |        Balance |
+----------------+--------------------------+----------------+----------------+
| Thu Aug 21 2008 | ...                     |       1,723.00 |       1,723.00 |
| Wed Dec 31 2008 | ...                     |  (       45.33)|       1,677.67 |
| Mon Jul 13 2009 | ...                     |      10,388.07 |      12,065.74 |
| Sun Jan 10 2010 | ...                     |  (      654.32)|      11,411.42 |
+----------------+--------------------------+----------------+----------------+
```

- The spec requires you to call `ctime()` to convert a Unix timestamp to string
  - then pick the right characters to display as date
  - e.g., `ctime()` returns `"Thu Aug 30 08:17:32 2012\n"`
    - becareful, `ctime()` returns a pointer that points to a *global variable*, so you must *make a copy*

```
char date[16];
char buf[26];
strncpy(buf, ctime(...), sizeof(buf));
date[0] = buf[0];
date[1] = buf[1];
...
date[15] = '\0';
```

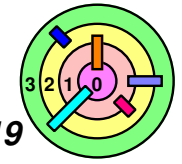


Sort Command

```
0000000001111111111222222222233333333334444444444555555555566666666667777777778
1234567890123456789012345678901234567890123456789012345678901234567890123456789
```

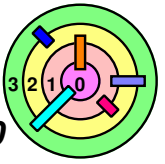
Date	Description	Amount	Balance
Thu Aug 21 2008	...	1,723.00	1,723.00
Wed Dec 31 2008	...	(45.33)	1,677.67
Mon Jul 13 2009	...	10,388.07	12,065.74
Sun Jan 10 2010	...	(654.32)	11,411.42

- Format your data in your own buffer
  - write a function to "format" numeric fields into null-terminated strings
    - it's a little more work, but you really should have this code isolated
      - in case you have bugs, just fix this function
  - you can even do the formatting when you append or insert your data structure to your list
    - need more fields in your data structure
  - this way, you can just print things out easily
  - use `printf("%s", ...)` to print a field to `stdout`



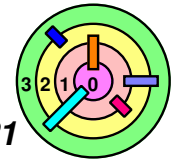
Warmup #1

- I'm giving you a lot of details on how to do things in C
  - this is the first and last assignment that I will do this!
  - you must learn C (and Unix) on your own
- Read man pages
- Ask questions in class Google Group
  - or send e-mail to me
- Come to office hours, especially if you are stuck



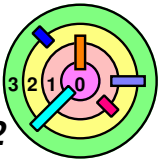
Some General Requirements

- Some major requirements for all programming assignments
  - severe pentalty for failing `make` (can lose up to 10 points)
    - we will attempt to fix your Makefile if we cannot compile your code
    - we are *not permitted to change your code*
  - severe pentalty for using large memory buffers
    - if input file is large, you must not read the whole file into into a large memory buffer
      - must learn how to read a large file properly
  - severe pentalty for any segmentation fault -- you must test your code well
  - severe pentalty for not using separate compilation or for having all your source code in header files -- you must learn to plan how to write your program
    - read warmup1 FAQ to see what's the best way to go about this



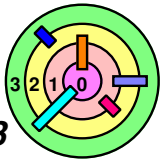
Grading Requirements

- For *warmup assignments*, it's important that every byte of your data is read and written correctly
  - you are not entitled to partial credit just because you wrote some code
- For *warmup assignments*, you should run your code against the *grading guidelines* on 32-bit Ubuntu 16.04
  - must not change the commands there
    - we will change the data for actual grading, but we will stick to the commands (as much as we can)
  - to be fair to all, running scripts in the grading guidelines on the grader's 32-bit Ubuntu 16.04 is *the only way we can grade*



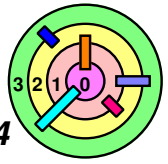
Separate Compilation

- Break up your code into *modules*
  - *compile the modules separately*, at least one rule per module per rule in the `Makefile`
  - a separate rule to *link* all the modules together
    - if your program requites additional libraries, add them to the link stage
- To receive full credit for separate compilation
  - to create an executable, at a minimum, you must run the compiler at least *twice* and the linker *once*
  - see the *warmup1 FAQ* for exactly how to avoid losing points



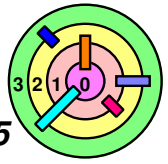
README

- Start with the README templates from the spec
  - *BUILD & RUN* (required)
    - replace "(Comments: ?)" with appropriate responses
  - *SELF-GRADING* (required)
    - replace each "?" with a *numerical score*
  - *BUGS / TESTS TO SKIP* (required)
    - replace "(Comments: ?)" with a list of tests to skip or "none"
      - you would still lose points, but this may prevent losing additional points in another part
  - *ADDITIONAL INFORMATION FOR GRADER* (optional)
    - grader must read this
  - *OTHERS* (optional)
    - will *not* be considered for grading
- There should be no "?" left in a response *in a required section* after you have filled out a README file correctly
  - 0.5 pt will be deducted if a "?" is not replaced with something appropriate or if the line is omitted



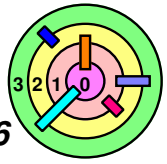
Code Design - Functional vs. Procedural

- Don’t design your program "procedurally"
- You need to learn how to write functions!
  - please note that this is not "functional programming" ("functional programming" is something else)
  - a function has a well-defined interface
    - what are the meaning of the parameters
    - what does it suppose to return
  - pre-conditions
    - what must be true when the function is entered
    - you assume that these are true
      - you can verify it if you want
  - post-conditions
    - what must be true when the function returns
  - you design your program by making designing a sequence of function calls



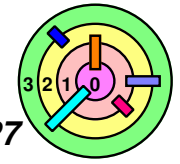
Warmup #1 - Miscellaneous Requirements

- Run your code against the *grading guidelines*
  - must not change the test program
- You must not use any *external code fragments*
- You must not use *array* to implement any list functions
  - *must use pointers* because this is a pointer exercise
  - read my *review about pointers* in warmup1 FAQ
- It's important that every byte of your data is read and written correctly.
  - `diff` commands in the grading guidelines must *not* produce *any* output or you will not get credit
    - what does "not produce any output" mean?
      - it means exactly what it says!
- Please see Warmup #1 spec for additional details
  - please read the *entire* spec (including the grading guidelines) *yourself*



Development

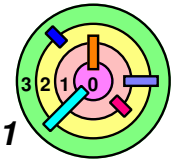
- Get familiar with *"Warmup #1 FAQ"* and *"Programming FAQ"*
- Text Editors
  - `emacs, pico, vi`
  - some students like Sublime Text
    - you are on your own with Sublime Text
- Compiler
  - `"gcc --version"` should say it's version 5.4.something
- IDE
  - some students like Eclipse
    - you are on your own with Eclipse
  - the grader is *not permitted* to use an IDE to compile or run your program
    - if you use an IDE, it's your responsibility to make sure that you provide a Makefile so that the grader can type the command in the spec to compile



Warmup #1 (Part 2)

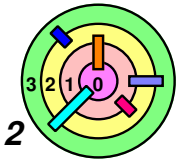
Bill Cheng

http://merlot.usc.edu/william/usc/



listtest

- Use provided `listtest.c` and `Makefile` to create `listtest`
  - `listtest` must run without error and you must not change `listtest.c` and `Makefile`
  - They specifies how your code is expected to be used
- You should learn how to run `listtest` under `gdb`

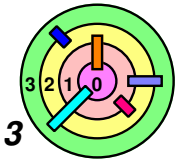


gdb listtest Exercise

- Do the following gdb exercise with listtest
  - *IMPORTANT:* draw picture on a piece of paper!
  - first, change "num_items=64" in DoTest() to "num_items=3"

```
make
gdb listtest
(gdb) break DoTest
(gdb) run
(gdb) n                          ← do this 6 times, you are now at call to CreateTestList()
(gdb) print &(list.anchor)       ← what's the address of the anchor?
(gdb) print list                 ← does the list look like an empty list?
(gdb) n                          ← returned from CreateTestList()
(gdb) print list                 ← does the list look like a 3-item list?
(gdb) print list.anchor          ← what's in the anchor?
(gdb) print *(list.anchor.next)
(gdb) print *(list.anchor.next->next)
(gdb) print *(list.anchor.next->next->next)
```

this should be the last list element,
does its next pointer point to the anchor?



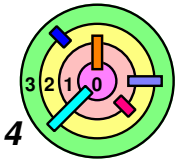
C Review: C-string Functions

- C-string manipulating functions in your kernel assignments (some do not assume C-string):

```
int    memcmp(const void *cs, const void *ct, size_t count);
void  *memcpy(void *dest, const void *src, size_t count);
int    strncmp(const char *cs, const char *ct, size_t count);
int    strcmp(const char *cs, const char *ct);
char  *strcpy(char *dest, const char *src);
char  *strncpy(char *dest, const char *src, size_t count);
void  *memset(void *s, int c, size_t count);
size_t strnlen(const char *s, size_t count);
size_t strlen(const char *s);
char  *strchr(const char *s, int c);
char  *strrchr(const char *s, int c);
char  *strstr(const char *s1, const char *s2);
char  *strcat(char *dest, const char *src);
char  *strdup(const char *s);
char  *strtok(char *s, const char *d);
```

  - you also need:

```
char  *fgets(char *s, int size, FILE *stream);
int   printf(const char *format, ...);
```



C File I/O Review: Reading Text Input

- Read in an entire line using `fgets()`
  - especially since we know the maximum line length
- If a filename is given, use `fopen()` to get a file pointer (`FILE*`)

```
FILE *fp = fopen(..., "r");
```

  - read man pages of `fopen()`
  - if a filename is not given, you will be reading from "standard input" (i.e., file descriptor 0)

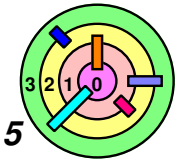
```
FILE *fp = stdin;
```

  - see grading guidelines for examples

```
cat ... | ./warmup1 sort
```

  - pass the file pointer around so that you run the same code whether you input comes from a file or `stdin`

```
My402List list;
if (!My402ListInit(&list)) { /* error */ }
if (!ReadInput(fp, &list)) { /* error */ }
if (fp != stdin) fclose(fp);
SortInput(&list);
PrintStatement(&list);
```



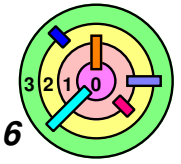
C File I/O Review: Parsing Text Input

- Read a line

```
char buf[2000];
if (fgets(buf, sizeof(buf), fp) == NULL) {
  /* end of file */
} else {
  /* parse it */
}
```

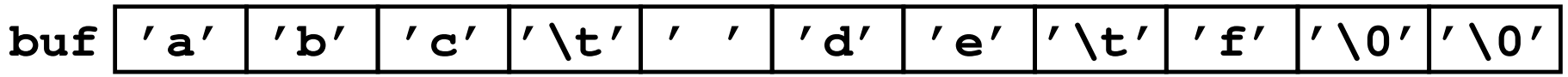
- Parse a line according to the spec
  - find an *easy* and *correct* way to parse the line
    - according to the spec, each line must have exactly 3 <TAB> characters
    - I think it's easy and correct to go after this

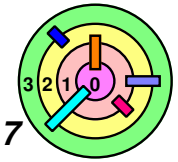
```
char *start_ptr = buf;
char *tab_ptr = strchr(start_ptr, '\t');
if (tab_ptr != NULL) {
  *tab_ptr++ = '\0';
}
/* start_ptr now contains a
   "null-terminated string" */
```



C File I/O Review: Parsing Text Input

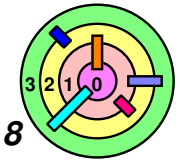
```
char *start_ptr = buf;
char *tab_ptr = strchr(start_ptr, '\t');
if (tab_ptr != NULL) {
  *tab_ptr++ = '\0';
}
/* start_ptr now contains a
   "null-terminated string" */
```





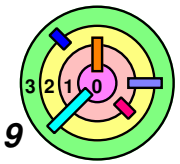
C File I/O Review: Parsing Text Input

```
char *start_ptr = buf;
char *tab_ptr = strchr(start_ptr, '\t');
if (tab_ptr != NULL) {
  *tab_ptr++ = '\0';
}
/* start_ptr now contains a
   "null-terminated string" */
```



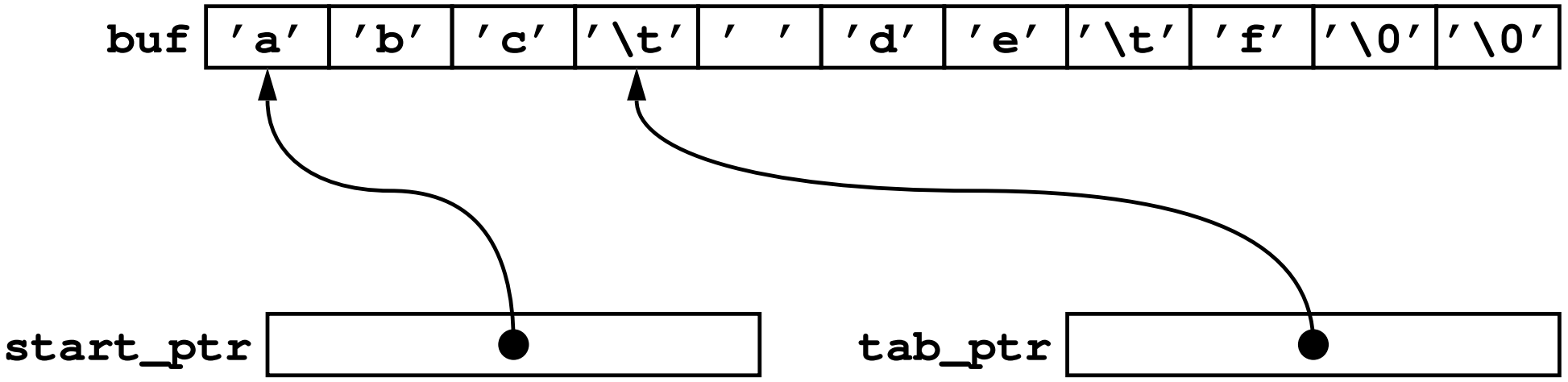
C File I/O Review: Parsing Text Input

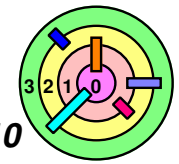
```
char *start_ptr = buf;
char *tab_ptr = strchr(start_ptr, '\t');
if (tab_ptr != NULL) {
  *tab_ptr++ = '\0';
}
/* start_ptr now contains a
   "null-terminated string" */
```



C File I/O Review: Parsing Text Input

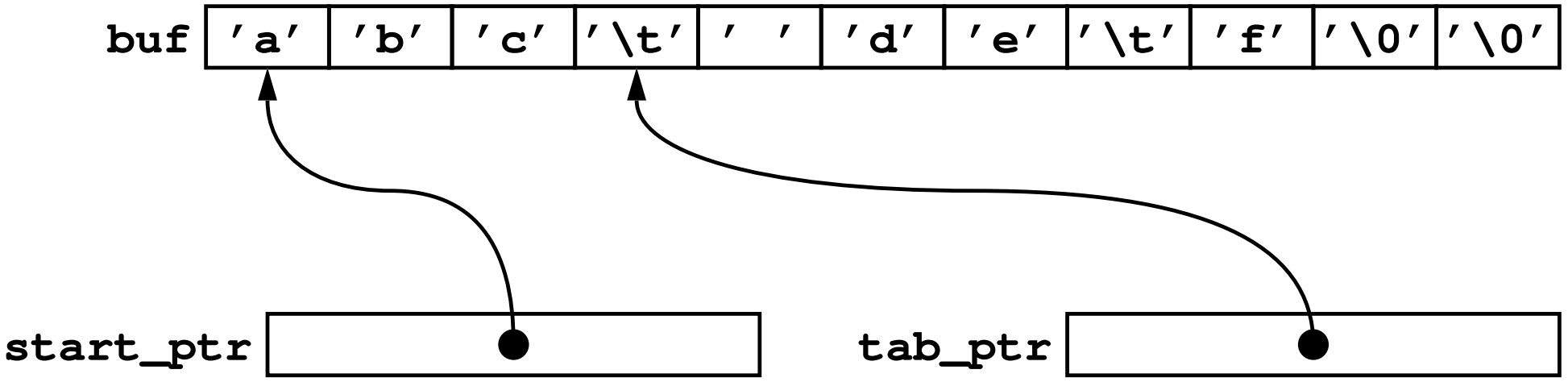
```
char *start_ptr = buf;
char *tab_ptr = strchr(start_ptr, '\t');
if (tab_ptr != NULL) {
  *tab_ptr++ = '\0';
}
/* start_ptr now contains a
   "null-terminated string" */
```

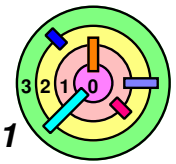




C File I/O Review: Parsing Text Input

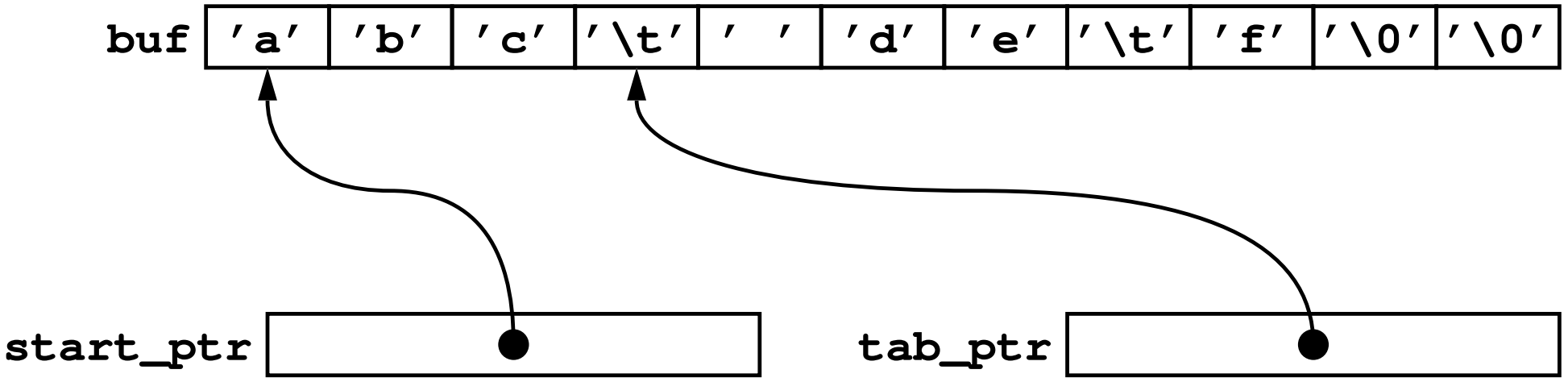
```
char *start_ptr = buf;
char *tab_ptr = strchr(start_ptr, '\t');
if (tab_ptr != NULL) {
  *tab_ptr++ = '\0';
}
/* start_ptr now contains a
   "null-terminated string" */
```

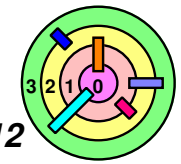




C File I/O Review: Parsing Text Input

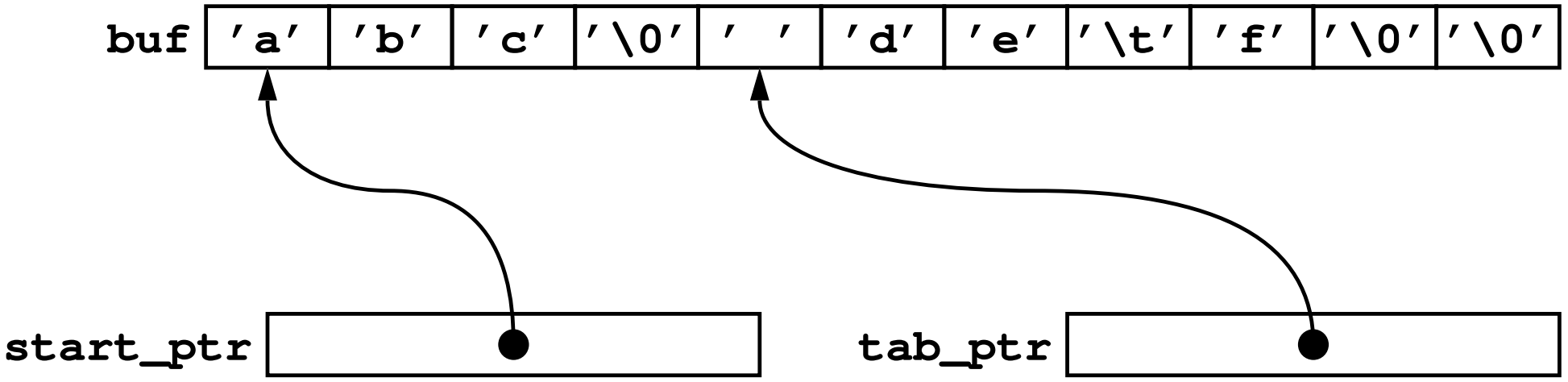
```
char *start_ptr = buf;
char *tab_ptr = strchr(start_ptr, '\t');
if (tab_ptr != NULL) {
  *tab_ptr++ = '\0';
}
/* start_ptr now contains a
   "null-terminated string" */
```

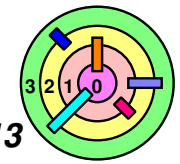




C File I/O Review: Parsing Text Input

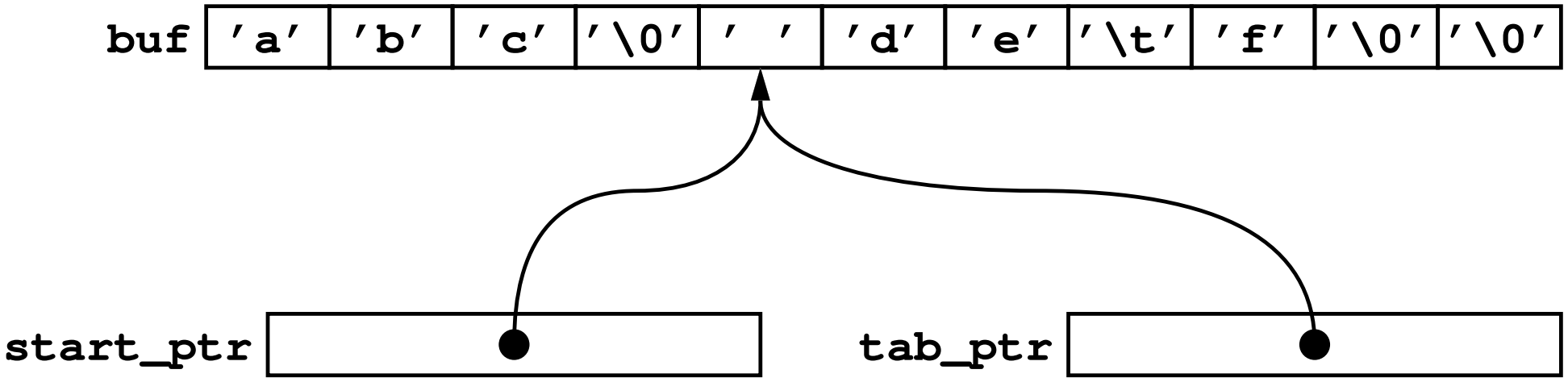
```
char *start_ptr = buf;
char *tab_ptr = strchr(start_ptr, '\t');
if (tab_ptr != NULL) {
  *tab_ptr++ = '\0';
}
/* start_ptr now contains a
   "null-terminated string" */
```

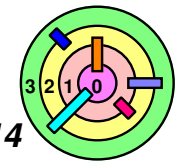




C File I/O Review: Parsing Text Input (2nd Iteration)

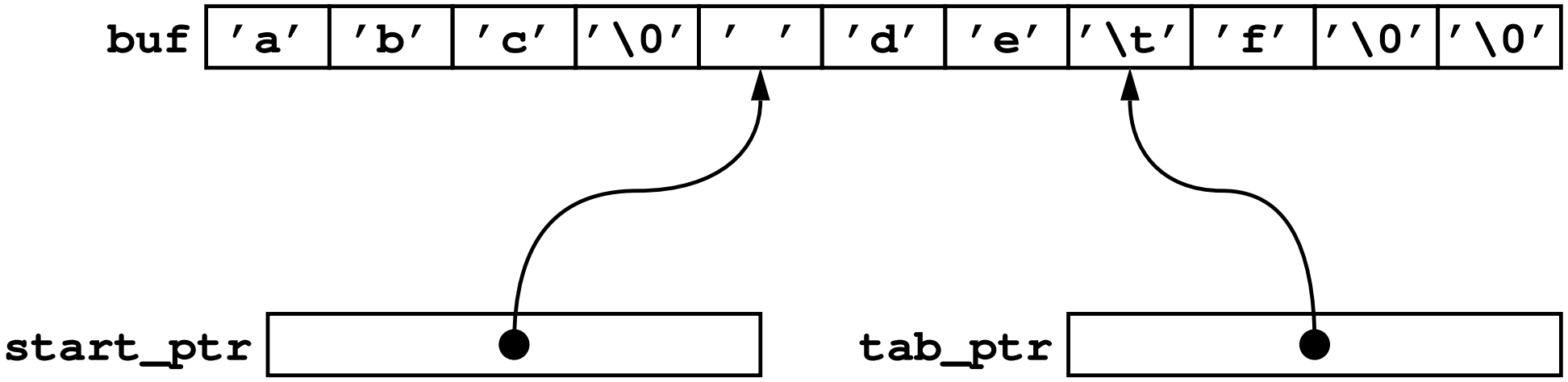
```
start_ptr = tab_ptr;
tab_ptr = strchr(start_ptr, '\t');
if (tab_ptr != NULL) {
  *tab_ptr++ = '\0';
}
/* start_ptr now contains a
   "null-terminated string" */
```

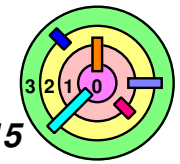




C File I/O Review: Parsing Text Input (2nd Iteration)

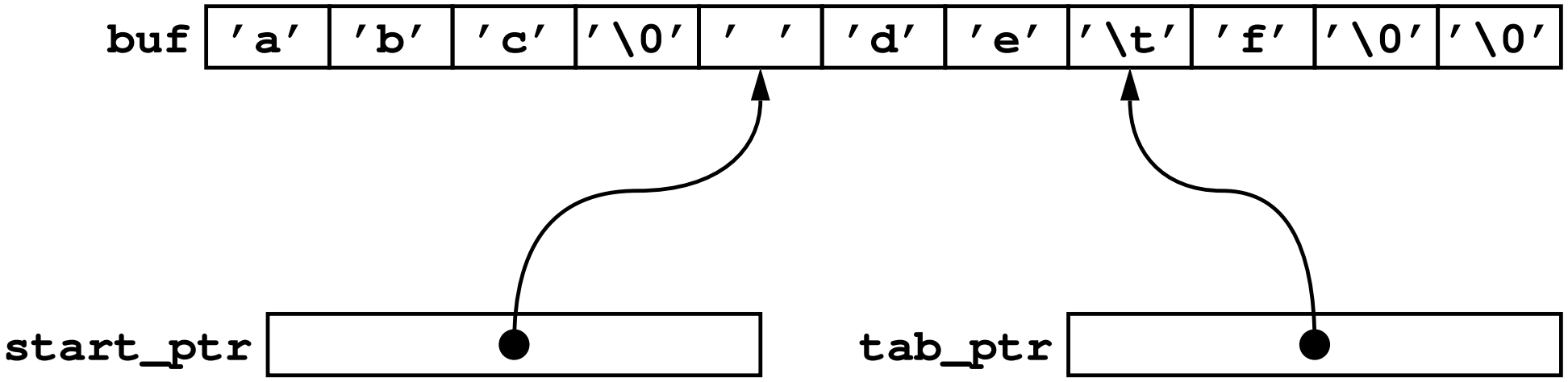
```
    start_ptr = tab_ptr;
    tab_ptr = strchr(start_ptr, '\t');
    if (tab_ptr != NULL) {
      *tab_ptr++ = '\0';
    }
    /* start_ptr now contains a
       "null-terminated string" */
```

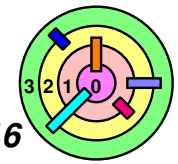




C File I/O Review: Parsing Text Input (2nd Iteration)

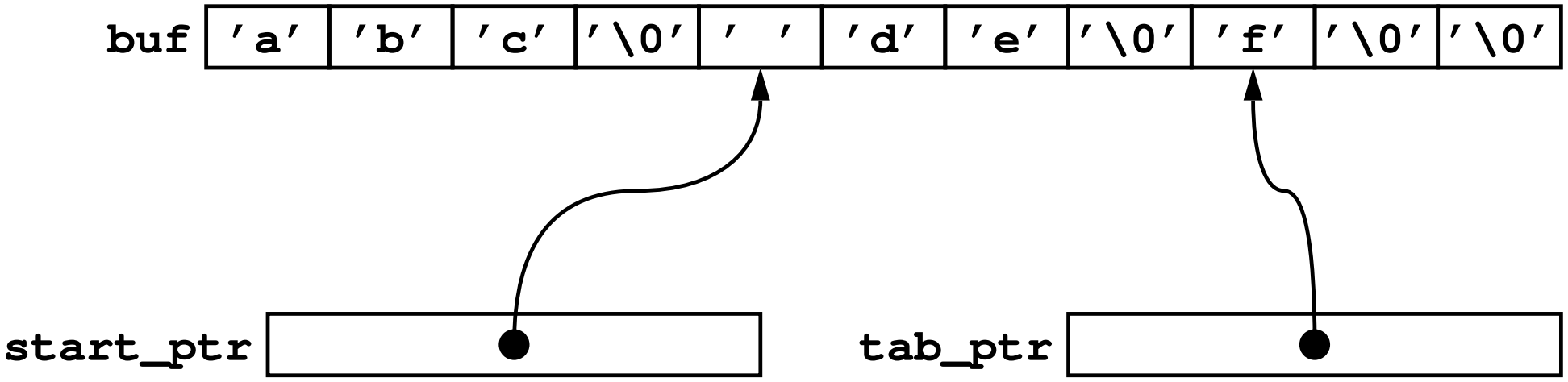
```
start_ptr = tab_ptr;
tab_ptr = strchr(start_ptr, '\t');
if (tab_ptr != NULL) {
  *tab_ptr++ = '\0';
}
/* start_ptr now contains a
   "null-terminated string" */
```

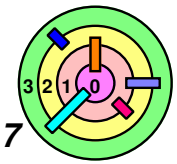




C File I/O Review: Parsing Text Input (2nd Iteration)

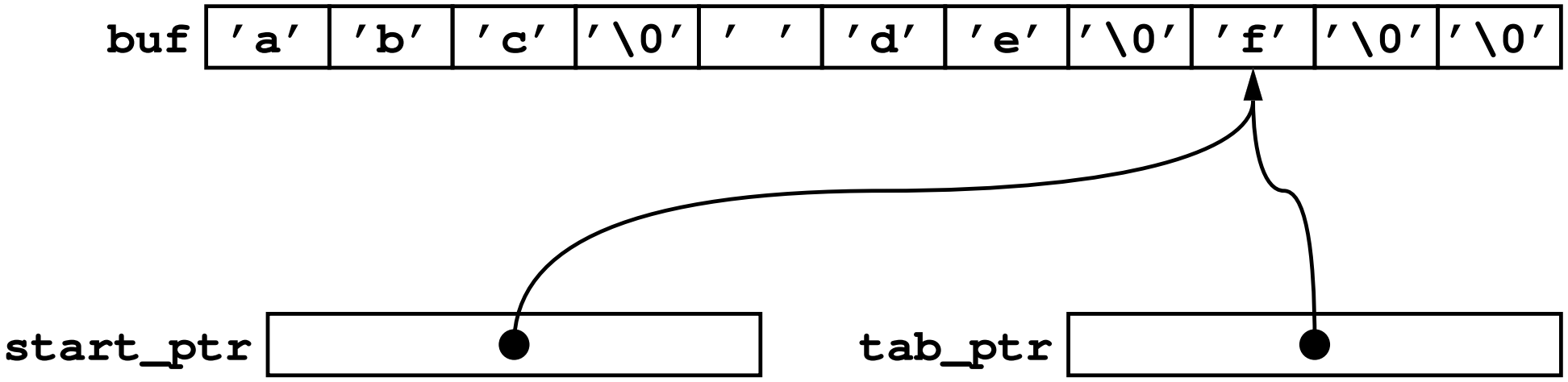
```
start_ptr = tab_ptr;
tab_ptr = strchr(start_ptr, '\t');
if (tab_ptr != NULL) {
  *tab_ptr++ = '\0';
}
/* start_ptr now contains a
   "null-terminated string" */
```

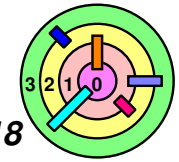




C File I/O Review: Parsing Text Input (3rd Iteration)

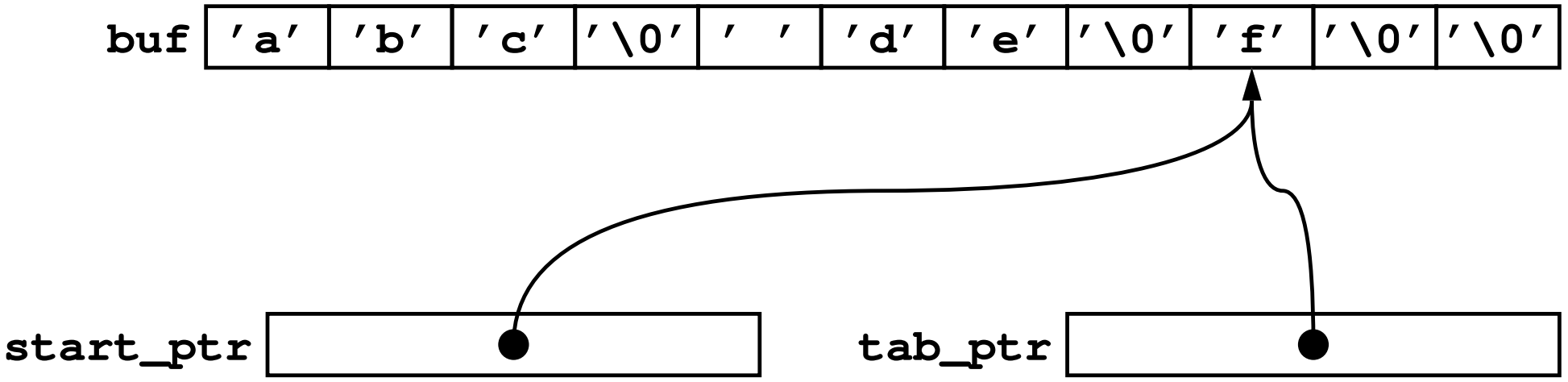
```
start_ptr = tab_ptr;
tab_ptr = strchr(start_ptr, '\t');
if (tab_ptr != NULL) {
  *tab_ptr++ = '\0';
}
/* start_ptr now contains a
   "null-terminated string" */
```

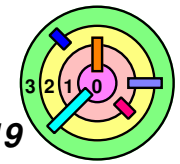




C File I/O Review: Parsing Text Input (3rd Iteration)

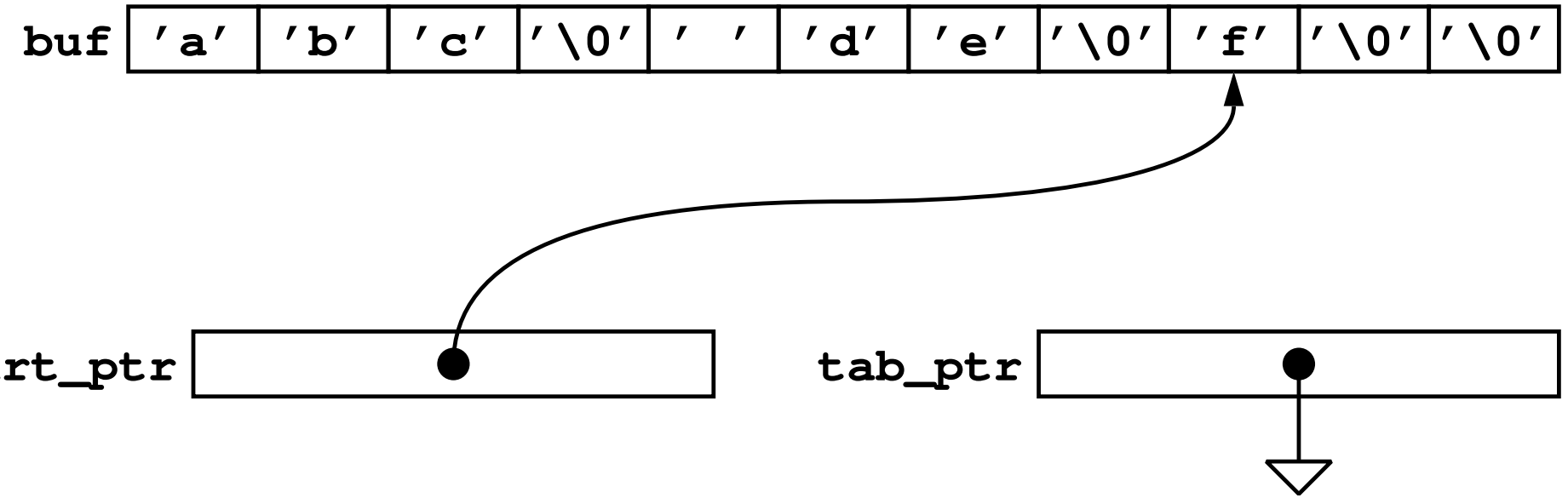
```
start_ptr = tab_ptr;
tab_ptr = strchr(start_ptr, '\t');
if (tab_ptr != NULL) {
  *tab_ptr++ = '\0';
}
/* start_ptr now contains a
   "null-terminated string" */
```





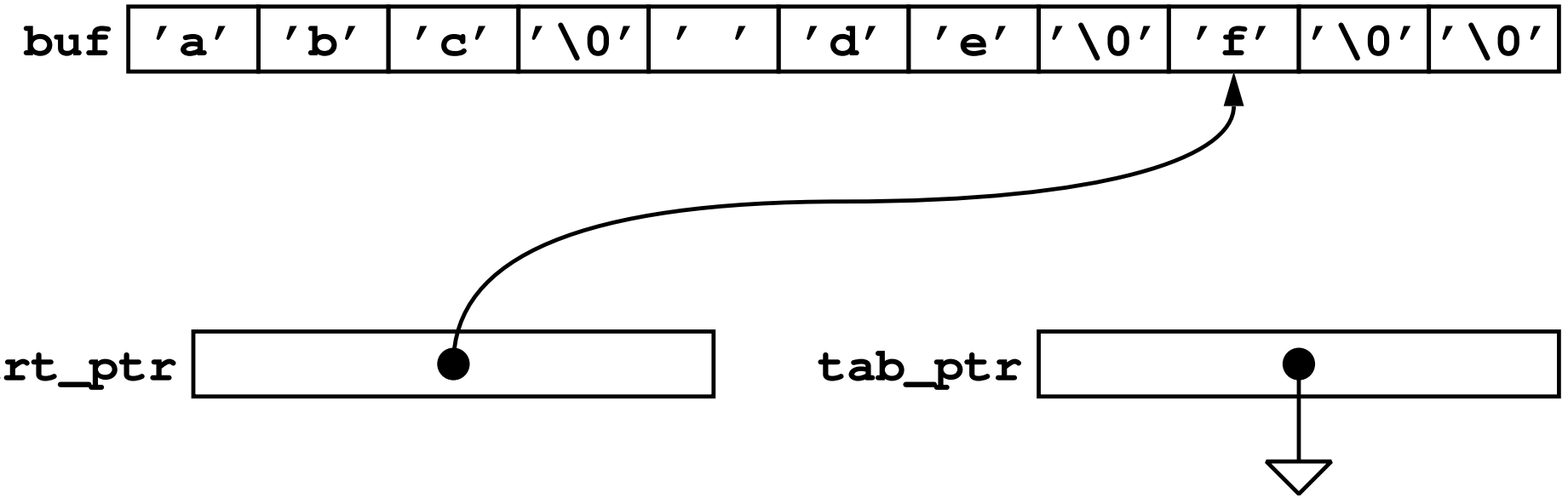
C File I/O Review: Parsing Text Input (3rd Iteration)

```
start_ptr = tab_ptr;
tab_ptr = strchr(start_ptr, '\t');
if (tab_ptr != NULL) {
  *tab_ptr++ = '\0';
}
/* start_ptr now contains a
   "null-terminated string" */
```



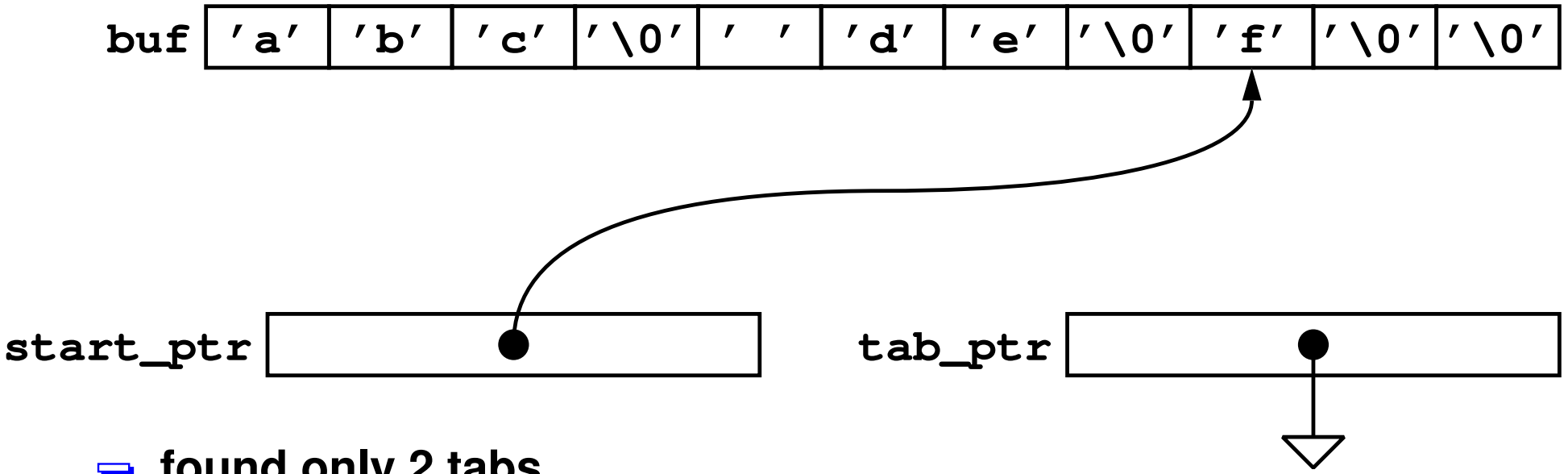
C File I/O Review: Parsing Text Input (3rd Iteration)

```
start_ptr = tab_ptr;
tab_ptr = strchr(start_ptr, '\t');
if (tab_ptr != NULL) {
  *tab_ptr++ = '\0';
}
/* start_ptr now contains a
   "null-terminated string" */
```

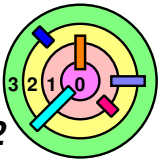


C File I/O Review: Parsing Text Input (3rd Iteration)

```
start_ptr = tab_ptr;
tab_ptr = strchr(start_ptr, '\t');
if (tab_ptr != NULL) {
  *tab_ptr++ = '\0';
}
/* start_ptr now contains a
   "null-terminated string" */
```



- found only 2 tabs
  - need to keep a count



C Function Review: Returning A C-String

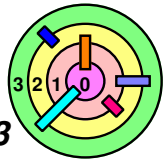
- How to return a C-string?
  - easiest way is to provide a buffer in the caller
    - can only do this if you know what buffer size to use

- Caller code (to get a C-string from callee):

```
char field[1024];

if (GetField(field, sizeof(field) > 0) {
  /* field now has a C-string */
} else {
  /* print error message */
  exit(-1);
}
```

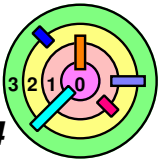
- Callee code:

```
int GetField(char *buf, int buf_sz)
{
  /* setup start_ptr to point to a field */
  strncpy(buf, start_ptr, buf_sz);
  buf[buf_sz-1] = '\0';
  return strlen(buf);
}
```



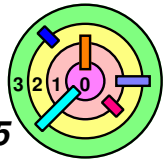
Warmup #1

- I'm giving you a lot of details on how to do things in C
  - this is the first and last assignment that I will do this!
  - you must learn C on your own
- Read man pages
- Ask questions in class Google Group
  - or send e-mail to me
- Come to office hours, especially if you are stuck



Warmup #1 - Miscellaneous Requirements

- Run your code against the *grading guidelines*
  - must not change the test program
- You must not use any *external code fragments*
- You must not use *array* to implement any list functions
  - must use pointers
- If input file is large, you must not read the whole file into into a large memory buffer
- It's important that every byte of your data is read and written correctly.
  - `diff` commands in the grading guidelines must *not* produce *any* output or you will not get credit
- Please see Warmup #1 spec for additional details
  - please read the *entire* spec *yourself*

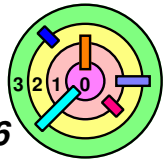


Warmup #1 - Miscellaneous Requirements

- Explain how `Makefile` work

```
listtest: listtest.o my402list.o
        gcc -o listtest -g listtest.o my402list.o

listtest.o: listtest.c my402list.h
        gcc -g -c -Wall listtest.c

my402list.o: my402list.c my402list.h
        gcc -g -c -Wall my402list.c

clean:
        rm -f *.o listtest
```

- Explain why the above `Makefile` satisfied ***separate compilation*** requirement
- Explain how to convert the above `Makefile` to a `Makefile` students can use for part (2)



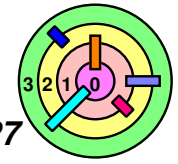
Demos

- If there is time in discussion section, demonstrate how create a `warmup1` subdirectory on 32-bit Ubuntu 16.04
  - if there is no time in discussion section, students should try all these on their own
- Create "hello.c" on 32-bit Ubuntu 16.04

```
#include <stdio.h>

int main(int argc, char *argv[])
{
  printf("Hello World!\n");
  return 0;
}
```

- Demonstrate how to run gcc

```
gcc -g -Wall hello.c
```



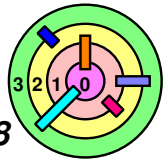
Demos

- Create "args.c" on 32-bit Ubuntu 16.04

```
#include <stdio.h>

static int foobar = 0x12345678;

int main(int argc, char *argv[])
{
  for (int i = 0; i < argc; i++) {
    printf("argv[%d] = '%s'\n", i, argv[i]);
  }
  return 0;
}
```

- Compile and run:

```
gcc -g -Wall args.c
./a.out x y z
```

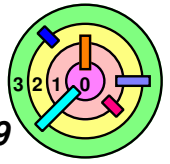


Demos

- Demonstrate how to debug `a.out`

```
gdb a.out
(gdb) break main
(gdb) run abc xyz
(gdb) print argc
(gdb) print argv[0]
(gdb) print argv[1]
(gdb) print argv[2]
(gdb) print &argc
(gdb) print &i
(gdb) print &foobar
(gdb) next
(gdb)
```

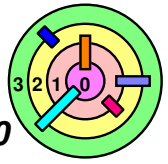
- Go to warmup1 spec and demonstrate how to copy and paste from the grading guidelines
- Go to warmup1 spec and demonstrate how to make a submission and get a PIN



Unix Commands

- Walk through and demonstrate the commands on the Unix Command Line Reference web page
  - click on the "summary of some commonly used Unix commands" link at the bottom of the class home page

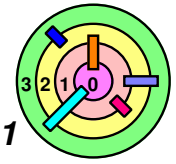
```
ls
cat
more
pwd
mkdir (directory name)
cd
cp (src file path) (dest file path)
mv (src file path) (dest file path)
man (cmd name)
rm (file path)
rmdir (empty directory name)
ps
kill (proc id)
pico (file path)
exit
```



Warmup #2 (Part 1)

Bill Cheng

http://merlot.usc.edu/william/usc/

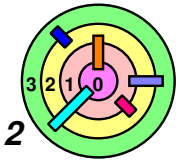


Multi-threading Exercise

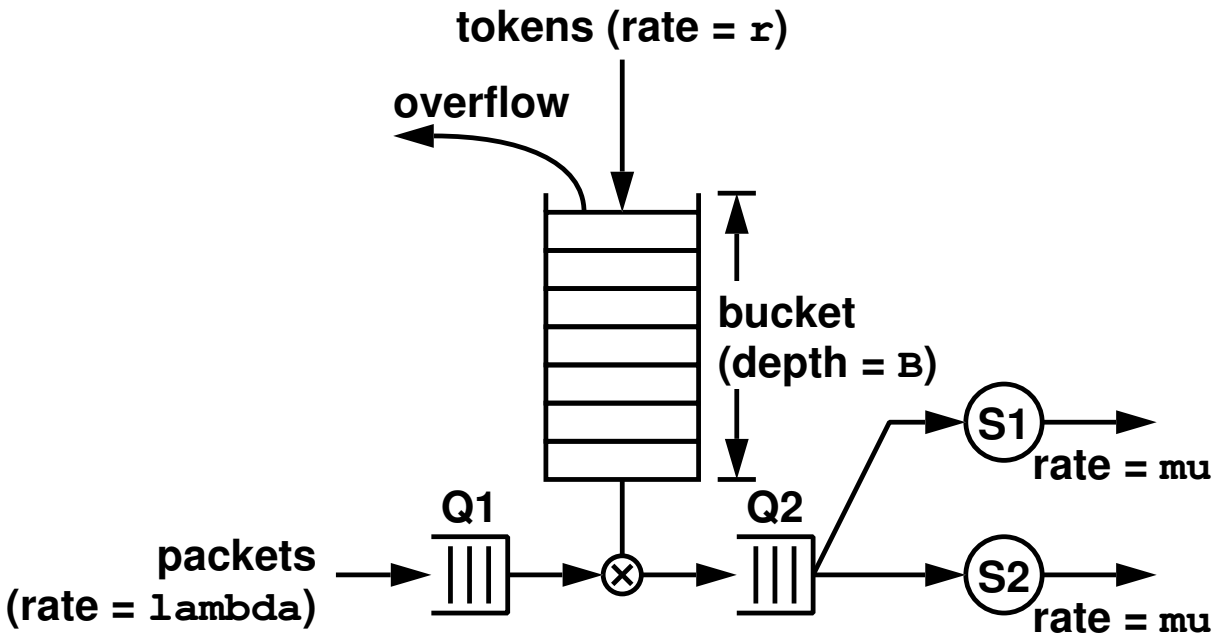
- Make sure you are familiar with the *pthreads* library
  - Ch 2 of textbook - threads, signals
    - additional resource is a book by Nichols, Buttlar, and Farrell *"Pthreads Programming"*, O'Rielly & Associates, 1996
  - you must learn how to use pthreads *mutex* and *condition variables* correctly
    - `pthread_mutex_lock()/pthread_mutex_unlock()`
    - `pthread_cond_wait()/pthread_cond_broadcast()`
      - do *not* use `pthread_cond_signal()` for warmup2

next week

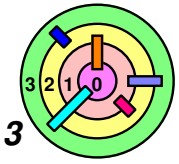
  - you must learn how to handle UNIX *signals* (<Ctrl+C>)
    - `sigprocmask()/sigwait()`
    - `pthread_cancel()`
  - you may want to learn how to disable/enable *cancellation* in pthreads
    - `pthread_setcancelstate()`



Token Bucket Filter

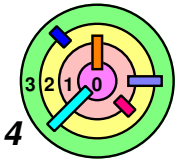


- Ex:
  - traffic controller/shaper
- Your job is to implement 4 *cooperating child threads* to move the packets along by following rules described in the spec
  - the main thread creates these threads, join with them, then print statistics



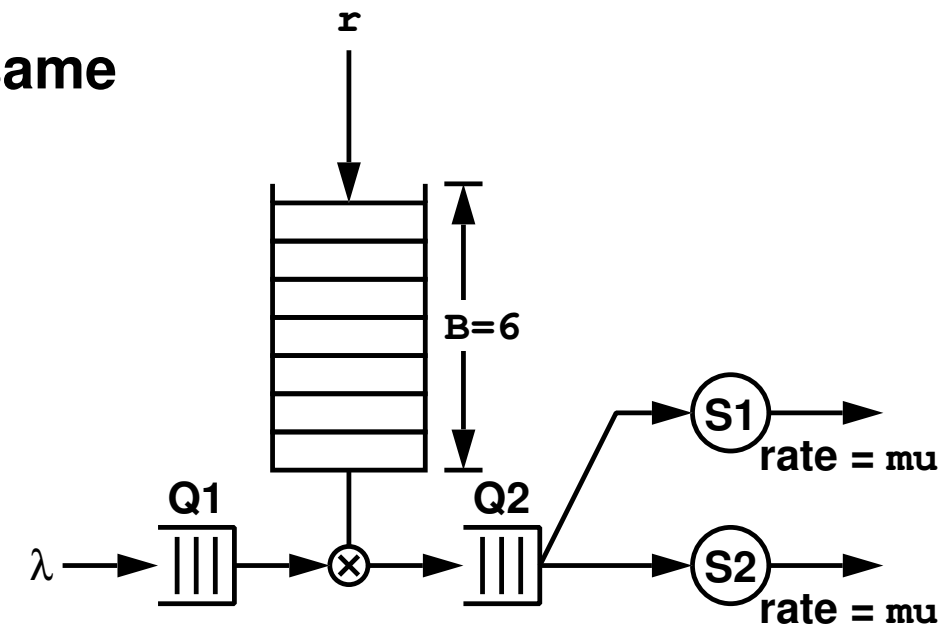
We Are *Not* Doing Event-driven Simulation

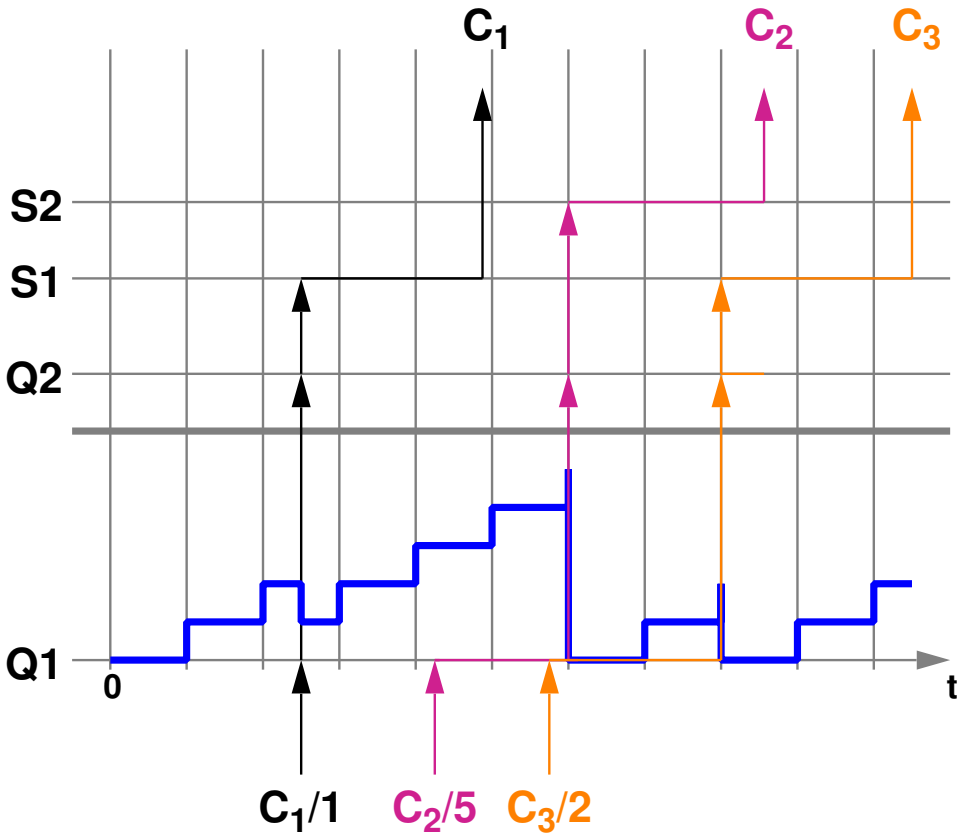
- An *event queue* is a sorted list of events according to timestamps; smallest timestamp at the head of queue
  - event has zero duration (events can happen at the same time)
- *Object oriented:* every object has a "next event" (what and when it will do next), this event is inserted into the event queue
- Execution: remove an event from the head of queue, "execute" the event (notify the corresponding object so it can insert the next event)
- Insert into the event queue according to timestamp of a new event; insertion may cause additional events to be deleted or inserted
- Potentially repeatable runs (if the same seed is used to initialize random number generator)
- The simulator never "sleeps"; it tries to run as fast as it can to finish the simulation as quickly as possible



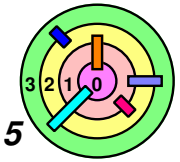
We Are *Not* Doing Event-driven Simulation

- Multiple event can happen at the same time in an *event-driven simulation*
  - we will *not* be doing that!



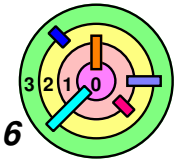


- $r_3 = d_3 - a_3$

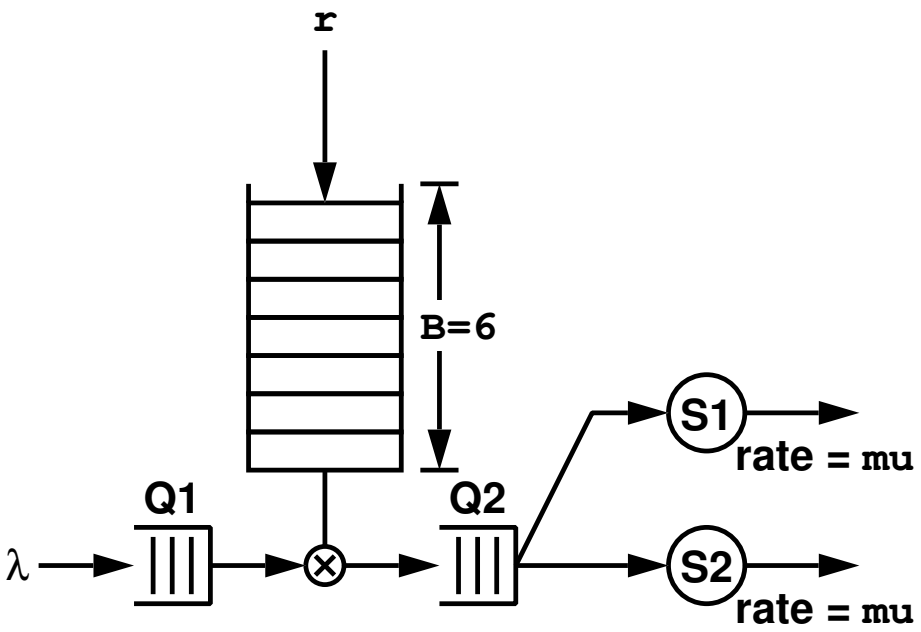


"Time Driven" Simulation

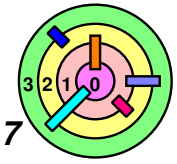
- We will use the words "simulation" and "emulation" interchangeably
- No "event queue"
  - every *active object* is implemented as a *thread*
  - threads interacting with one another through the use of *shared variables*
    - how else can threads "talk" to each other?!
- It takes time to execute simulation code
  - the time it takes to do all that is part of the simulation
  - to simulation the passing of time, call `usleep()`
    - e.g., if doing something takes `x` usec, call `usleep(x)`
    - Ubuntu does not run a "realtime" OS, it's "best effort"
    - `usleep(x)` will return more than `x` usec later
      - and sometimes, *a lot more* than `x` usec later
      - you need to decide if the extra delay is reasonable or it's due to a bug in your code



"Time Driven" Simulation

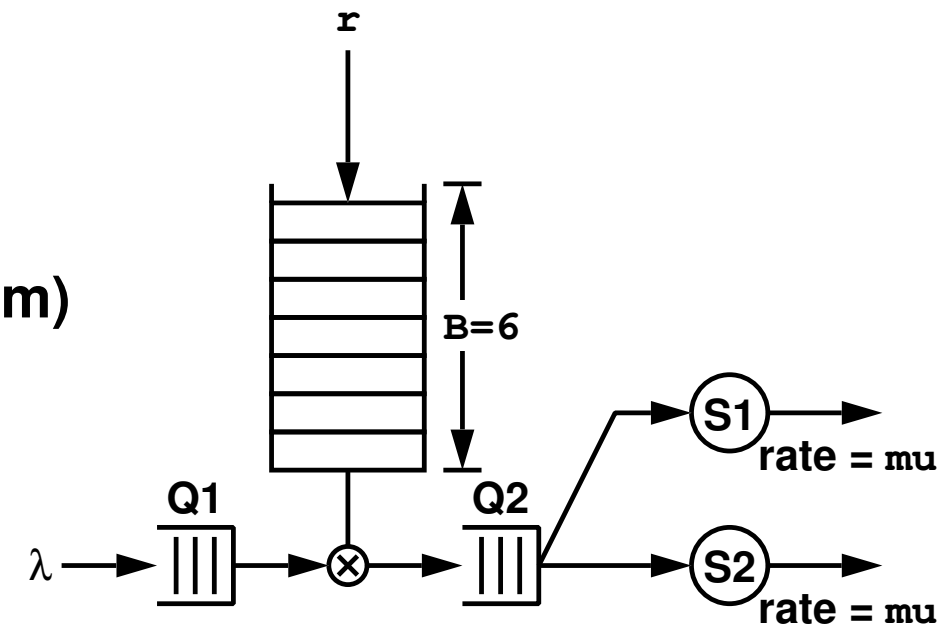


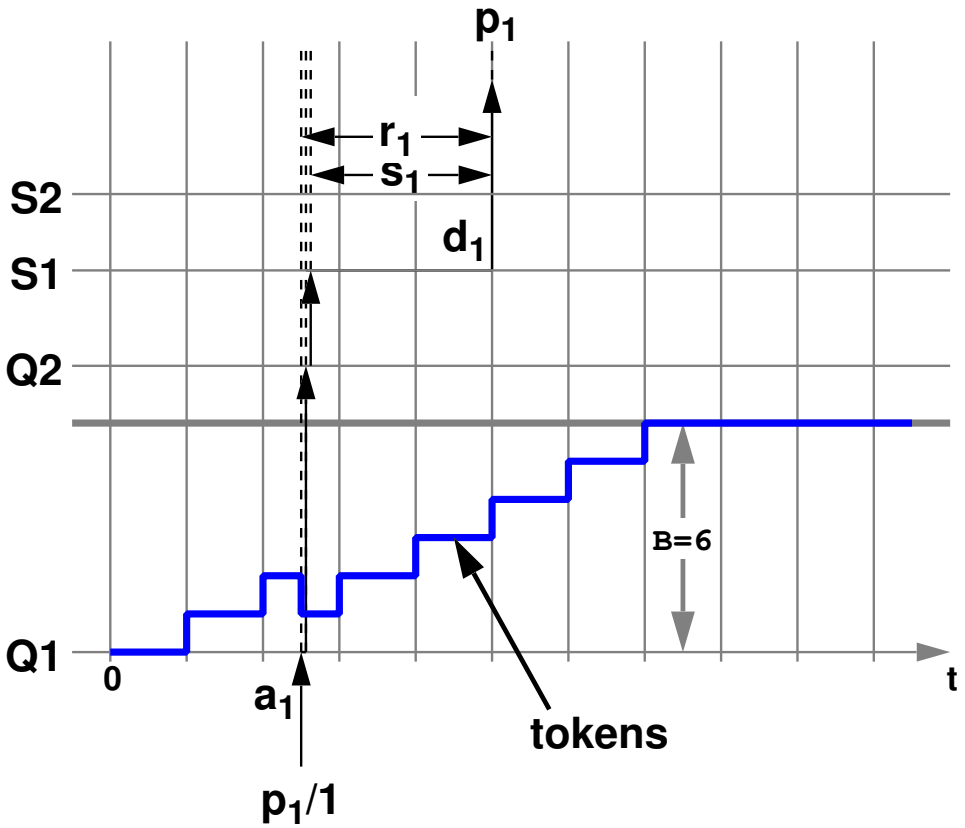
- Let your machine decide which thread to run next
  - results can never be reproducible exactly
  - debugging can be more challenging
- Compete for resources (such as Q1, Q2, and anything shared), must use a *single mutex*
- *No busy-waiting*
  - must use a *single CV*



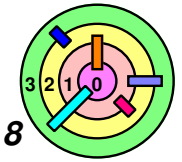
Arrivals & Departures

- a_i : arrival time
- d_i : departure time
- s_i : service time
- r_i : response time (time in system)
- q_i^1, q_i^2 : queueing/waiting time



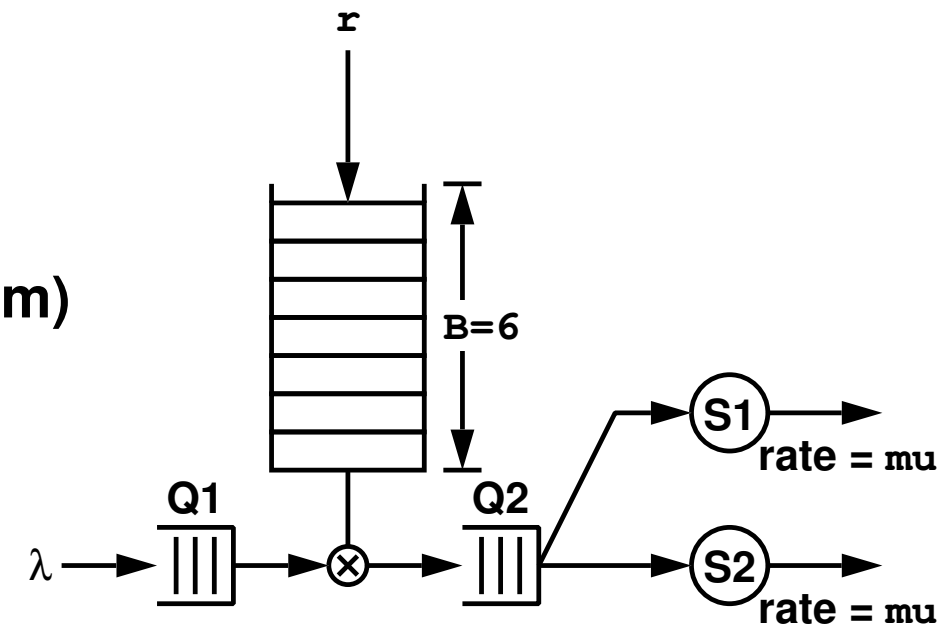


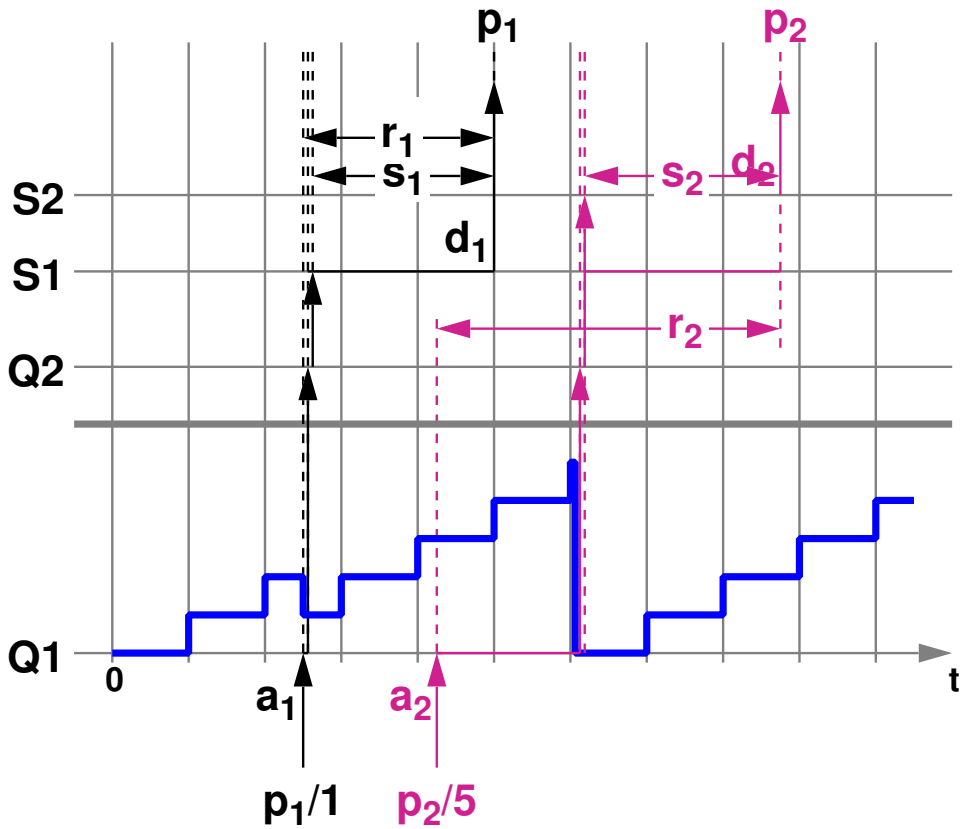
- $r_1 = d_1 - a_1$



Arrivals & Departures

- a_i : arrival time
- d_i : departure time
- s_i : service time
- r_i : response time (time in system)
- q_i^1, q_i^2 : queueing/waiting time

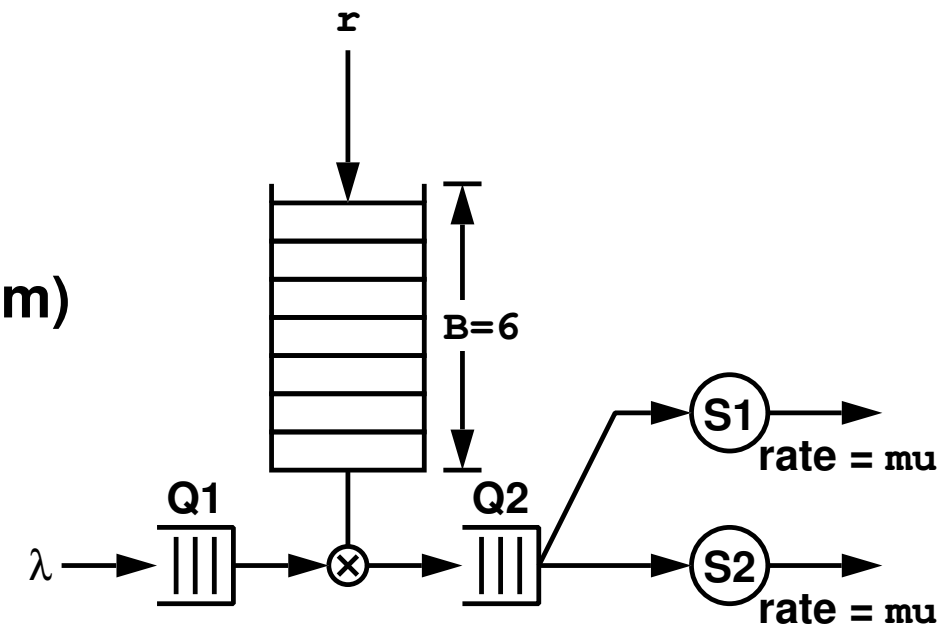


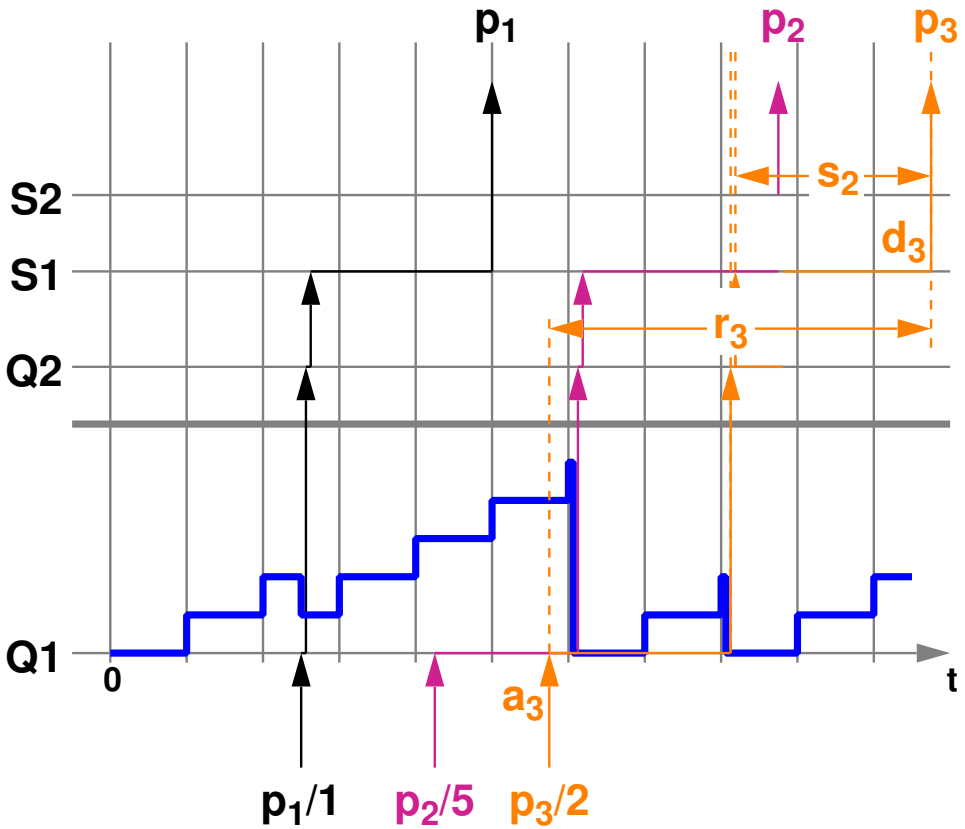


- $r_2 = d_2 - a_2$

Arrivals & Departures

- a_i : arrival time
- d_i : departure time
- s_i : service time
- r_i : response time (time in system)
- q_i^1, q_i^2 : queueing/waiting time

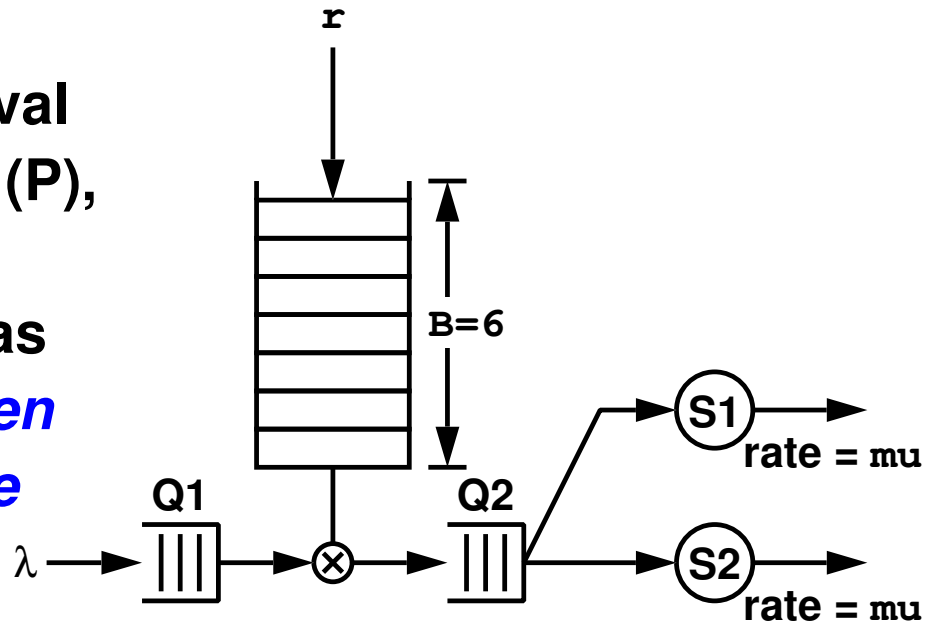




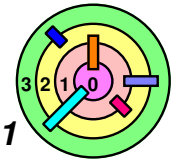
- $r_3 = d_3 - a_3$

Simulation/Emulation

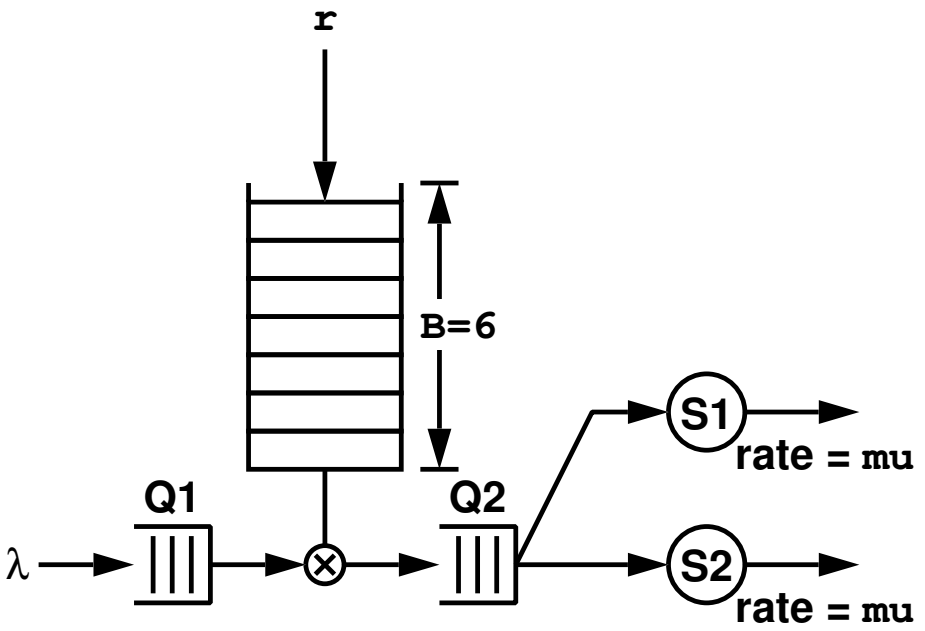
- Two simulation modes
  1) *Deterministic:* fixed inter-arrival time (1/λ), token requirement (P), and service time (1/mu)
  2) *Trace-driven:* every packet has its own *inter-arrival time*, *token requirement*, and *service time* (a line in a "`tsfile`")



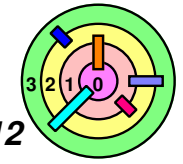
  - if you think about it carefully, there is really no difference between these two modes
    - write your code for the trace-driven mode
    - if running in deterministic mode, instead of reading a line from the "`tsfile`" to create a packet, just create a packet using information stored in global variables



Simulation/Emulation



- You will need to implement 4 *cooperating child threads*
  - *packet* arrival thread
  - *token* depositing thread
  - two *server* threads
  - these threads work together to simulate the operation of this token bucket filter
    - threads work together using *shared variables*



Simulation/Emulation

- Very high level pseudo-code for the *packet/token thread*:

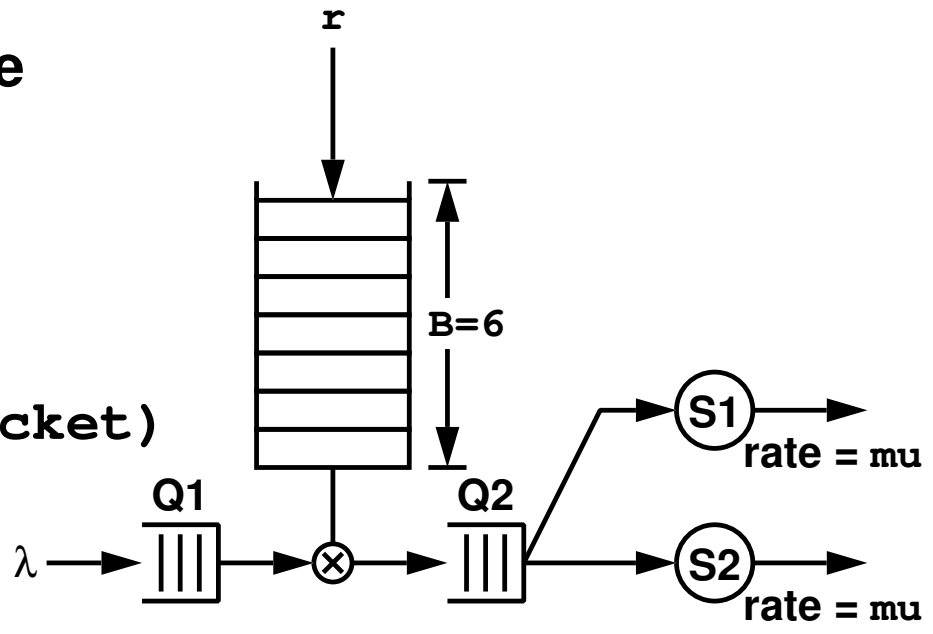
```
for (;;) {
  sleep
  generate a packet/token
  add packet/token to token bucket filter
}
```

  - where must you lock and unlock mutex?

- Very high level pseudo-code for the *server thread*:

```
for (;;) {
  wait for packet in Q2
  remove packet from Q2
  sleep (to transmit packet)
}
```

  - where must you lock and unlock mutex?

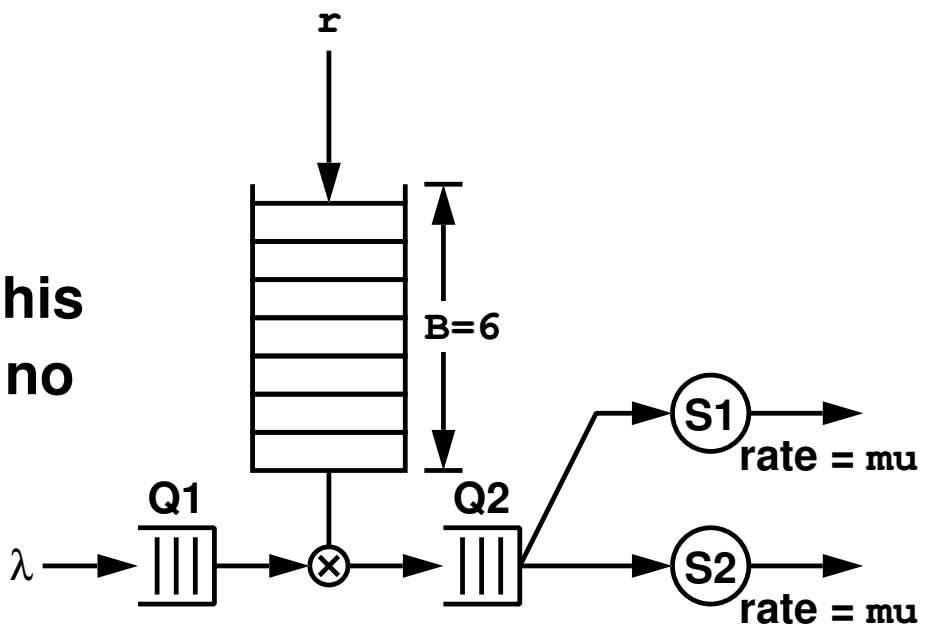


Simulation/Emulation

Packet thread pseudo-code (incomplete):

```
for (;;) {
  /* read a line from tsfile if in trace mode */
  get inter_arrival_time, tokens_needed, and service_time;
  /* calculate sleep time from inter_arrival_time */
  usleep(...);
  packet = NewPacket(tokens_needed, service_time, ...);
  pthread_mutex_lock(&mutex);
  Q1.enqueue(packet);
  ... /* other stuff */
  pthread_cond_broadcast(&cv);
  pthread_mutex_unlock(&mutex);
}
```

- must self-terminate as soon as this thread is no longer needed (i.e., no need to generate packets)
- must not call `pthread_cond_signal()`

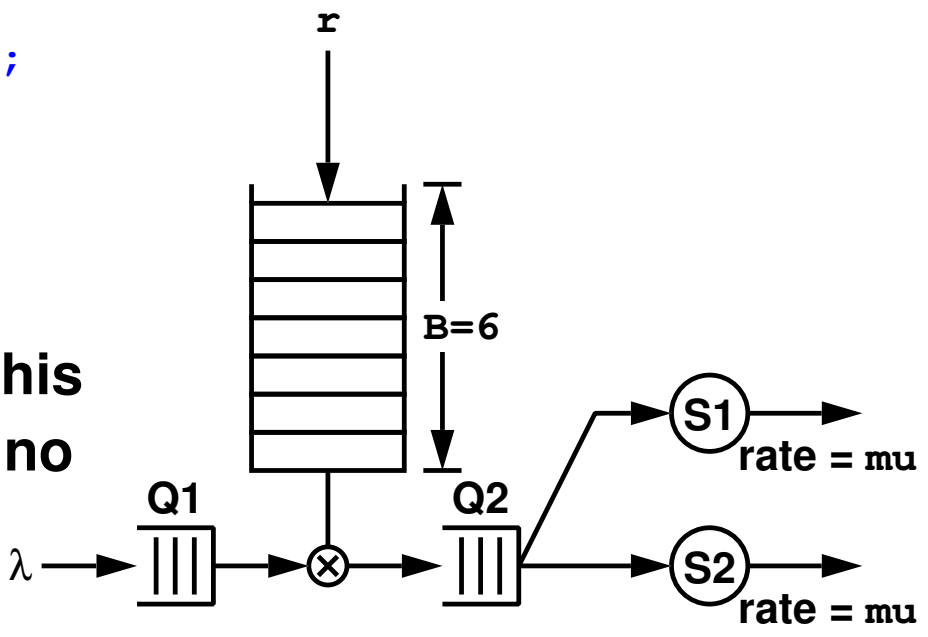


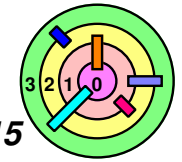
Simulation/Emulation

Token thread pseudo-code (incomplete):

```
for (;;) {
  /* calculate sleep time from inter-token arrival time */
  usleep(...);
  pthread_mutex_lock(&mutex);
  tokens++;
  if (first packet in Q1 can now be moved into Q2) {
    packet = Q1.dequeue();
    Q2.enqueue(packet);
    pthread_cond_broadcast(&cv);
    tokens = 0; /* why? */
  }
  pthread_mutex_unlock(&mutex);
}
```

- must self-terminate as soon as this thread is no longer needed (i.e., no need to generate tokens)
- must not call `pthread_cond_signal()`



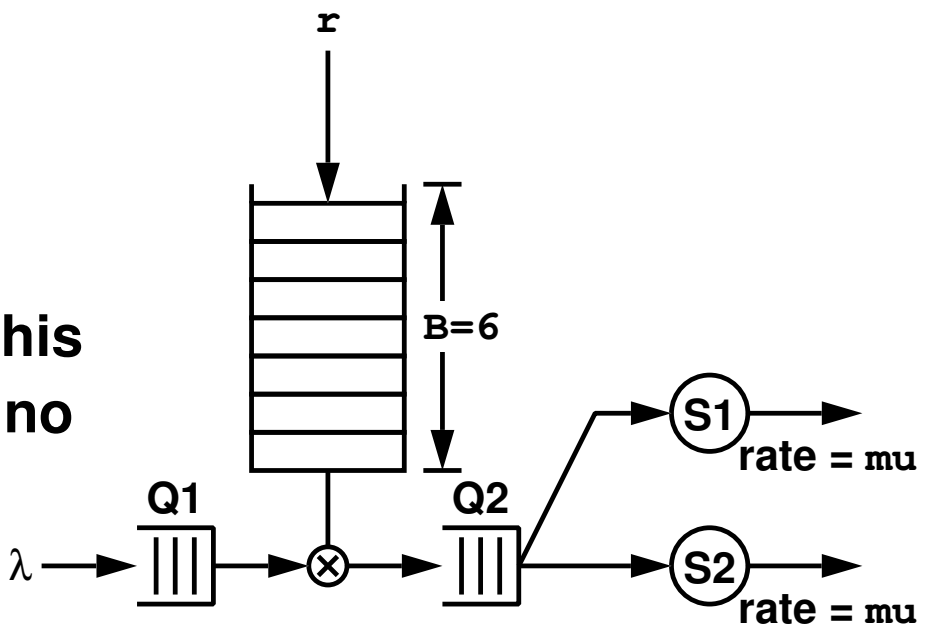


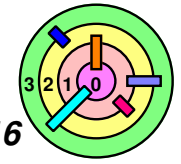
Simulation/Emulation

- Server threads pseudo-code (incomplete):
  - same first procedure for both server threads

```
for (;;) {
  /* wait for work */
  pthread_mutex_lock(&mutex);
  while (Q2.length() == 0 && !time_to_quit) {
    pthread_cond_wait(&cv, &mutex);
  }
  packet = Q2.dequeue();
  pthread_mutex_unlock(&mutex);
  /* work */
  usleep(packet.service_time);
}
```

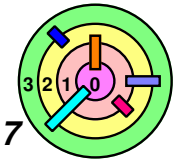
  - must self-terminate as soon as this thread is no longer needed (i.e., no need to transmit packets)





Simulation/Emulation

- Many other requirements, for example:
  - must move a packet at the correct time
    - if a packet is eligible to be moved from Q1 to Q2, it must happen immediately
  - all threads must self-terminate when they are no longer needed
  - drop packets
    - if the token requirement for an arriving packet is too large (i.e., > B), must drop the packet
  - drop tokens
    - if an arriving token finds a full bucket, the token is dropped
  - and many more...
    - please read the spec yourself (don't get it from classmates)

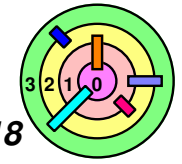


Program Printout

- Program output must look like what's in the spec
  - you must *NOT* wait for emulation to end to print all these

```
Emulation Parameters:
    number to arrive = 20
    lambda = 2              (if -t is not specified)
    mu = 0.35               (if -t is not specified)
    r = 4
    B = 10
    P = 3                   (if -t is not specified)
    tsfile = FILENAME           (if -t is specified)

00000000.000ms: emulation begins
00000251.726ms: token t1 arrives, token bucket now has 1 token
00000502.031ms: token t2 arrives, token bucket now has 2 tokens
00000503.112ms: p1 arrives, needs 3 tokens, inter-arrival time = 503.112ms
00000503.376ms: p1 enters Q1
00000751.148ms: token t3 arrives, token bucket now has 3 tokens
00000751.186ms: p1 leaves Q1, time in Q1 = 247.810ms, token bucket now has 0 token
00000752.716ms: p1 enters Q2
00000752.932ms: p1 leaves Q2, time in Q2 = 0.216ms
00000752.982ms: p1 begins service at S1, requesting 2850ms of service
00001004.271ms: p2 arrives, needs 3 tokens, inter-arrival time = 501.159ms
00001004.526ms: p2 enters Q1
00001007.615ms: token t4 arrives, token bucket now has 1 token
00001251.259ms: token t5 arrives, token bucket now has 2 tokens
00001505.986ms: p3 arrives, needs 3 tokens, inter-arrival time = 501.715ms
00001506.713ms: p3 enters Q1
00001507.552ms: token t6 arrives, token bucket now has 3 tokens
00001508.281ms: p2 leaves Q1, time in Q1 = 503.755ms, token bucket now has 0 token
00001508.761ms: p2 enters Q2
00001508.874ms: p2 leaves Q2, time in Q2 = 0.113ms
00001508.895ms: p2 begins service at S2, requesting 1900ms of service
...
```

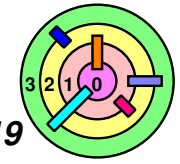
from commandline or `tsfile`



Program Printout

```
...
00003427.557ms: p2 departs from S2, service time = 1918.662ms, time in system = 2423.286ms
00003612.843ms: p1 departs from S1, service time = 2859.861ms, time in system = 3109.731ms
...
????????.???ms: p20 departs from S?, service time = ???.???ms, time in system = ???.???ms
????????.???ms: emulation ends

Statistics:

    average packet inter-arrival time = <real-value>
    average packet service time = <real-value>

    average number of packets in Q1 = <real-value>
    average number of packets in Q2 = <real-value>
    average number of packets at S1 = <real-value>
    average number of packets at S2 = <real-value>

    average time a packet spent in system = <real-value>
    standard deviation for time spent in system = <real-value>

    token drop probability = <real-value>
    packet drop probability = <real-value>
```

- ***Timestamps*** in the left column must have ***microsecond*** resolution
  - ***measured time interval*** must have ***microsecond*** resolution
  - use "`%.6g`" in `printf()` for `<real-value>`
- A value anywhere in the right column must be the exact differences between two corresponding timestamps



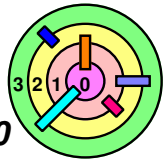
Program Printout

```
...
00003427.557ms: p2 departs from S2, service time = 1918.662ms, time in system = 2423.286ms
00003612.843ms: p1 departs from S1, service time = 2859.861ms, time in system = 3109.731ms
...
????????.???ms: p20 departs from S?, service time = ???.???ms, time in system = ???.???ms
????????.???ms: emulation ends

Statistics:

    average packet inter-arrival time = <real-value>
    average packet service time = <real-value>

    average number of packets in Q1 = <real-value>
    average number of packets in Q2 = <real-value>
    average number of packets at S1 = <real-value>
    average number of packets at S2 = <real-value>

    average time a packet spent in system = <real-value>
    standard deviation for time spent in system = <real-value>

    token drop probability = <real-value>
    packet drop probability = <real-value>
```

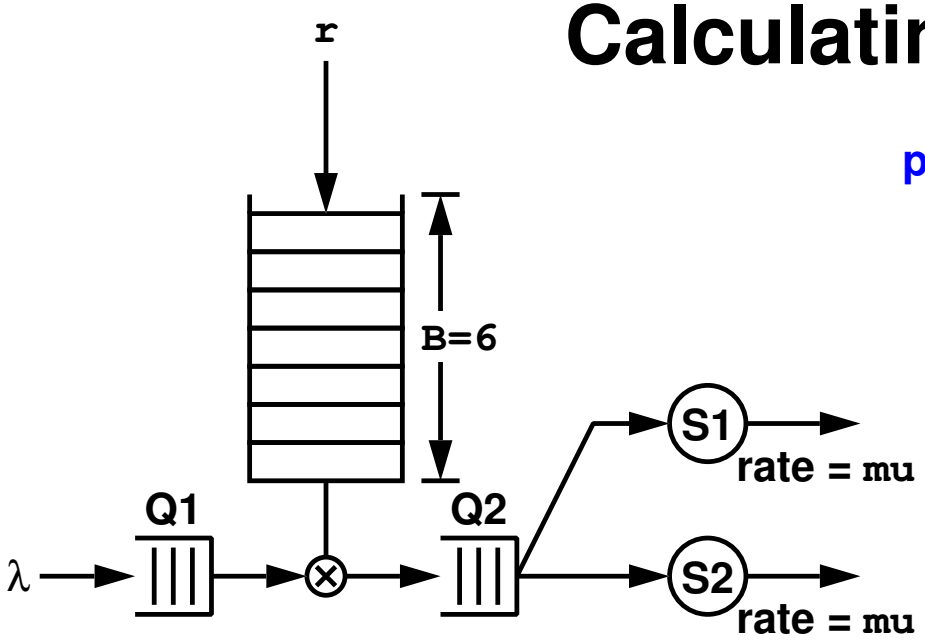
Ex: why must the service time for p1 be exactly 2859.861ms?

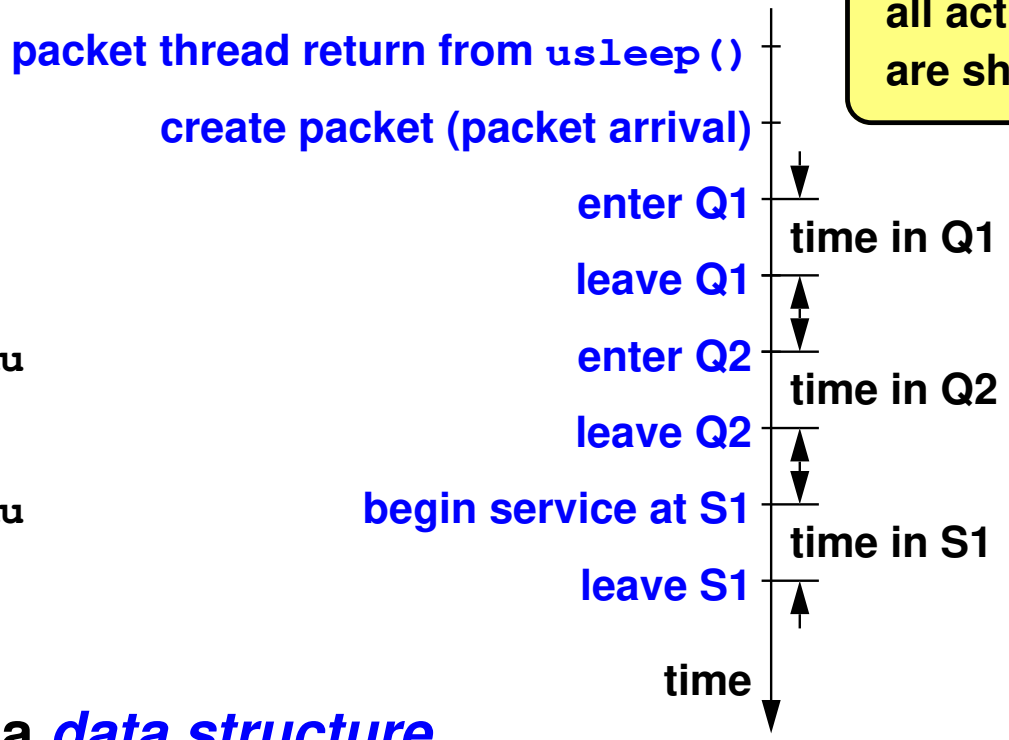
```
3612.843ms - 752.982ms = 2859.861ms
```

- why so strict?
  - your *printout* must be *self-consistent*



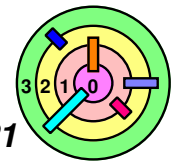
Calculating Statistics





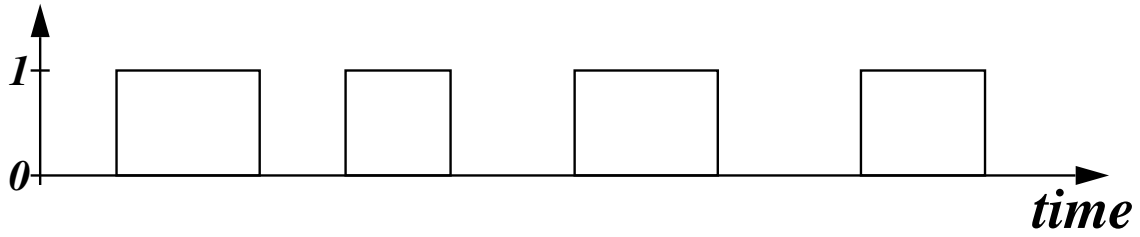
NOTE: **not all activities are shown**

- **A *packet* is not a thread, it's a *data structure***
  - **it should have 7 timestamps to store "measured" information**
  - **it should also store "packet specification" (such as specified inter-arrival time, token requirement, service time)**
    - **these are *not* "measured" values**
- **Some packets needs to be excluded from certain statistics**
  - **add to corresponding statistics only when a packet is being ejected**

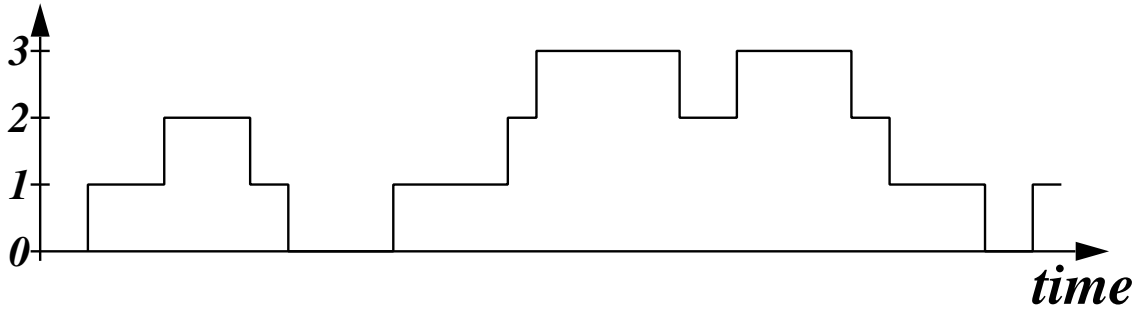


Mean and Standard Deviation

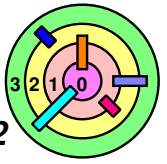
- Average time
  - for *n* samples, add up all the time and divide by *n*
- Average number of packets at a server
  - same a fraction of time the server is busy



- Average number of packets at Q1



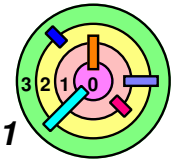
- Standard deviation is the squareroot of variance
  - $Var[X] = E[X^2] - (E[X])^2$
    - must use the *population variance* equation



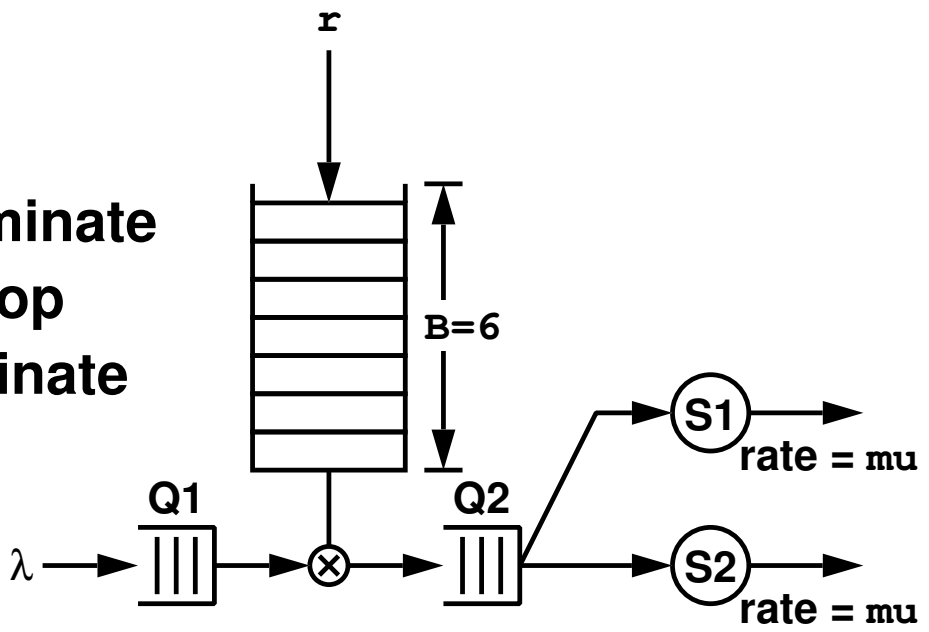
Warmup #2 (Part 2)

Bill Cheng

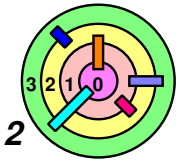
http://merlot.usc.edu/william/usc/



Handling <Ctrl+C> In Warmup #2

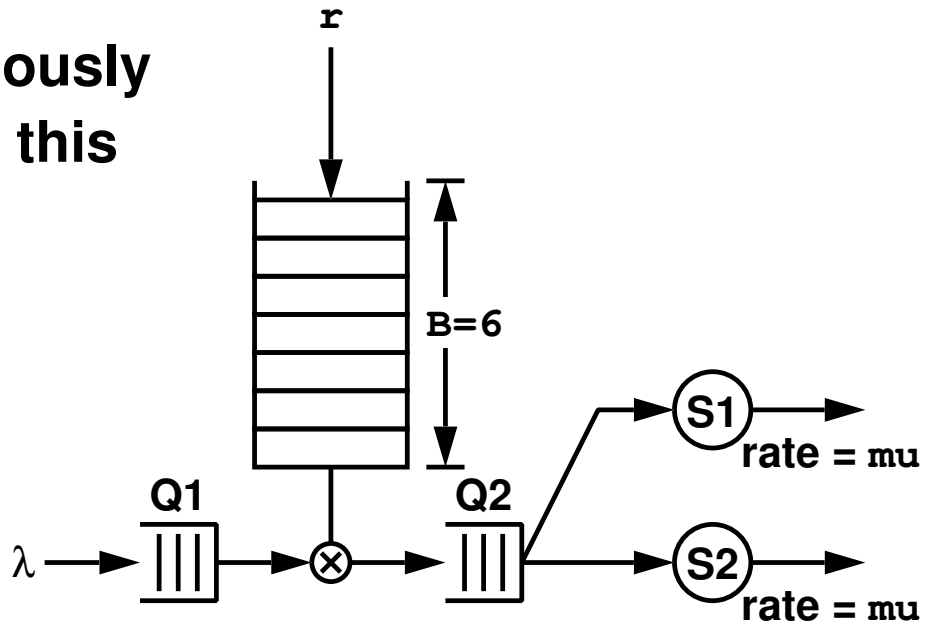


- <Ctrl+C>
  - packet arrival thread must stop generating packets and self-terminate
  - token depositing thread must stop generating tokens and self-terminate
  - a server thread must finish serving its current packet then self-terminate
  - no packets or tokens must arrive
  - need to take care of left-over packets that are stuck in Q1 and Q2
    - these are called *"removed packets"* because your simulator must remove them before you end your simulation
    - must distinguish 3 types of packets:
      1) *completed* packets
      2) *dropped* packets (packets that need > B tokens)
      3) *removed* packets

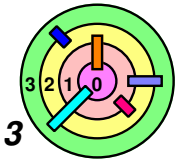


Handling <Ctrl+C> In Warmup #2

- You can catch <Ctrl+C> asynchronously
  - see lecture and understand why this is not a good idea
  - our recommendation is to *handle signals synchronously*
    - use a *signal-catching thread* and `sigwait()`
      - no signal handler at all



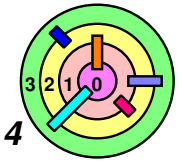
  - when <Ctrl+C> is caught, you should use the *pthread cancellation* mechanism to cancel packet and token threads
    - understanding thread cancellation will hopefully prevent you from make some mistakes in kernel assignment #1
      - although kernel cancellation is somewhat different from pthread cancellation



Designate A Thread To Catch A Signal

- Look at the man pages of `pthread_sigmask()` on Ubuntu 16.04
  - designate child thread to handler SIGINT
  - parent thread blocks SIGINT (some code deleted and use SIGINT instead of SIGQUIT):

```
#include <pthread.h>

int
main(int argc, char *argv[])
{
    pthread_t thread;
    sigset_t set;
    int s;

    sigemptyset(&set);
    sigaddset(&set, SIGINT);
    sigaddset(&set, SIGUSR1);
    s = pthread_sigmask(SIG_BLOCK, &set, NULL);
    if (s != 0)
        handle_error_en(s, "pthread_sigmask");

    s = pthread_create(&thread, NULL, &sig_thread, (void *) &set);
    if (s != 0)
        handle_error_en(s, "pthread_create");

    pause();
}
```



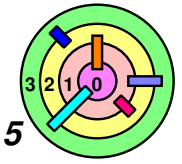
pthread_sigmask()

- Child thread example
  - child thread unblocks SIGINT

```
static void *
sig_thread(void *arg)
{
    sigset_t *set = arg;
    int s, sig;

    for (;;) {
        s = sigwait(set, &sig);
        if (s != 0)
            handle_error_en(s, "sigwait");
        printf("Signal handling thread got signal %d\n", sig);
    }
}
```

  - child thread is designated to handle SIGINT and SIGUSR1, no other thread will get SIGINT amd SIGUSR1

- Compile this code and run it
  - press <Ctrl+C> to see that this program cannot be killed with <Ctrl+C>
  - use "`kill -15`" command to kill this program

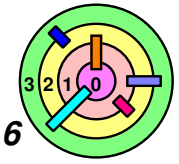


Example In Lecture

```
some_state_t state;
sigset_t set;

main() {
  pthread_t thread;
  sigemptyset(&set);
  sigaddset(&set,
            SIGINT);
  sigprocmask(
      SIG_BLOCK,
      &set, 0);
  // main thread
  //      blocks SIGINT
  pthread_create(
      &thread, 0,
      monitor, 0);
  long_running_proc();
}
```

```
void long_running_proc() {
  while (a_long_time) {
    pthread_mutex_lock(&m);
    update_state(&state);
    pthread_mutex_unlock(&m);
    compute_more();
  }
}

void *monitor() {
  int sig;
  while (1) {
    sigwait(&set, &sig);
    pthread_mutex_lock(&m);
    display(&state);
    pthread_mutex_unlock(&m);
  }
  return(0);
}
```

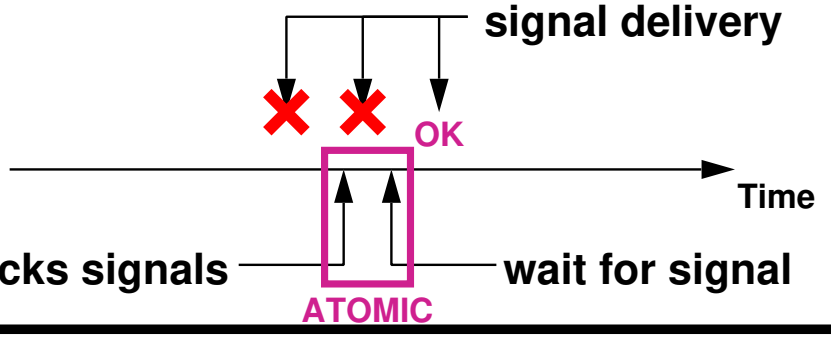
- this is the ***PREFERRED*** way to catch SIGINT for warmup2



sigwait()

```
int sigwait(sigset_t *set, int *sig)
```

- `sigwait()` blocks until a signal specified in `set` is received
  - return which signal caused it to return in `sig`
  - if you have a signal handler specified for `sig`, it will *not* get invoked when the signal is delivered
    - instead, `sigwait()` will return
- You should make sure that all the threads in your process have these signals blocked!
  - this way, when `sigwait()` is called, the calling thread temporarily becomes the *only* thread in the process who can receive the signal
- `sigwait(set)` *atomically unblocks* signals specified in `set` and *waits* for signal delivery



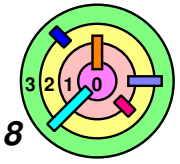
Warmup2 Catching <Ctrl+C> Made Easy

- Use a *<Ctrl+C>-catching thread* and *block SIGINT everywhere else*
  - when `sigwait()` returns because <Ctrl+C> has been pressed:

```
lock mutex
set global flag /* time to quit gracefully */
cancel packet and token threads
broadcast CV
unlock mutex
self terminate
```

    - according to spec, you must not cancel server threads

- Remember, if you don't use a <Ctrl+C>-catching thread, your regular threads can be borrowed to deliver signals
  - your code must deal with that and that can get very messy

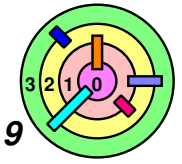


Warmup2 Cancellation Made Easy

- Only packet arrival and token depositing threads are allowed to be canceled
  - use a <Ctrl+C>-catching thread (i.e., use `sigwait()`)
  - handling clean up with `pthread_cleanup_push()` and `pthread_cleanup_pop()` can be messy
    - it's good to use them in general, but for warmup2, maybe we should try to make our lives easier by observing this:
    - first procedure of packet or token thread looks like:

```
do-forever
  sleep();
  lock mutex
  /* stuff */
  unlock mutex
end-do
```

    - if we have only one cancellation point, we know exactly where our threads will act on cancel

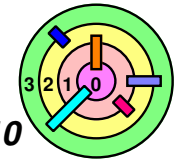


Warmup2 Cancellation Made Easy

- what if we change it to:

```
disable cancellation
do-forever
  enable cancellation
  sleep();
  disable cancellation
  lock mutex
  /* stuff */
  unlock mutex
end-do
```

- where can this thread act on cancel?
- would this make our thread *"unresponsive"* when <Ctrl+C> is pressed?
- there is a race condition in this code
  - what if the SIGINT-catching thread cancels this thread right when this thread is waiting for mutex lock?
  - must not create a new packet/token once you have printed the "SIGINT caught" message



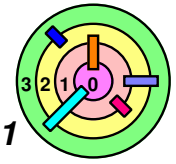
Statistics - Running Average

- Do not store all transmitted packets in a list and calculate statistics at the end of your simulation
  - it's a waste of memory
    - useful to learn how to write code with *small memory footprint*
  - keep a *running average* instead
    - just need one `int` and one `double`
      - `n` is the number of samples
      - `avg` is the average
      - when you get a new sample value, simply do:

```
avg = (n * avg + sample_value) / (n + 1)
n = n + 1
```

  - `avg` above is `E[X]`, how do you calculate $E[X^2]$?

```
avg2 = (n * avg2 + sample_value^2) / (n + 1)
n = n + 1
```

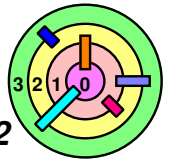


Statistics - Running Average

- can you use running averages for all these statistics?

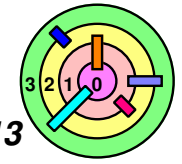
```
average packet inter-arrival time = <real-value>
average packet service time = <real-value>

average number of packets in Q1 = <real-value>
average number of packets in Q2 = <real-value>
average number of packets at S1 = <real-value>
average number of packets at S2 = <real-value>

average time a packet spent in system = <real-value>
standard deviation for time spent in system = <real-value>

token drop probability = <real-value>
packet drop probability = <real-value>
```

  - what should correspond to a sample value?
  - what types of packet must you consider?
    - avg packet inter-arrival time must consider ***all*** packets
    - other averages must only consider ***completed*** packets
    - packet drop probability is # dropped / # arrived



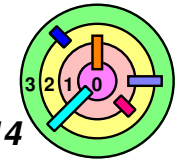
gdb

- When you get a breakpoint in `gdb`, what happened with other threads?
- If you break inside a signal handler and you type "`where`", how come `gdb` seems confused?



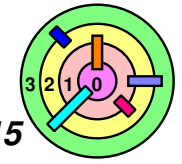
How To Learn New Concepts

- If there is a new concept that you are not familiar with, don't just try to write the final program
  - you won't know where the bugs are because you may not be clear about the concepts at multiple places
- Try writing small programs to test out ideas
  - try one idea at a time
  - use the debugger to get a better understanding of what's going on
  - then compile multiple ideas into one program and see if it works
- Ex:
  - `fork-wait.c`
  - `cat.c`
  - `redirect.c`
  - `thr-term.c`
  - `busywait.c, join.c`
  - `deadlock.c, trylock.c`
  - `whoopee.c`
  - `status-update.c`
  - `sigblock.c, sigwait.c`
  - `direct.c`
  - `cancel.c`
  - `three-threads.c`



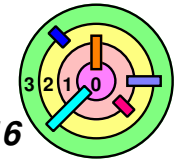
defs.h

```
#ifndef _DEFS_H_
#define _DEFS_H_

#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <signal.h>
#include <errno.h>

#include <pthread.h>

#ifndef NULL
#define NULL 0L
#endif /* ~NULL */

#endif /*_DEFS_H_*/
```



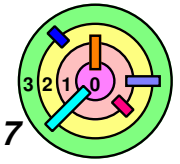
fork-wait.c

```
#include "defs.h"
#define NUM_CHILD 5
int sleep_time[NUM_CHILD], chd_num=0;
int main(int argc, char *argv[])
{
  srand48(time(0));
  for (chd_num=0; chd_num < NUM_CHILD; chd_num++) {
    sleep_time[chd_num] = lrand48() % 5000000;
    if (fork() == 0) {
      int pid=((int)getpid());
      printf("(Child) pid = %1d (0x%08x)\n", pid, pid);
      usleep(sleep_time[chd_num]);
      exit(child_pid+1);
    }
  }
  for (;;) {
    int pid=0, rc=0;
    if ((pid=wait(&rc)) == (-1)) break;
    printf("child %1d exited: 0x%08x.\n", pid, rc);
  }
  return 0;
}
```



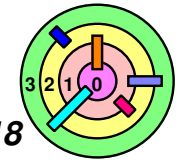
cat.c

```
#include "defs.h"
#define BUFSIZE 1024
int main(int argc, char *argv[])
{
  char buf[BUFSIZE];
  int n=0;
  const char *note="Write failed\n";

  while ((n = read(0, buf, sizeof(buf))) > 0)
    if (write(1, buf, n) != n) {
      (void)write(2, note, strlen(note));
      exit(1);
    }
  return 0;
}
```



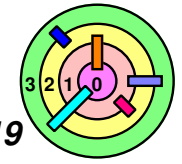
redirect.c

```
#include "defs.h"
int main(int argc, char *argv[])
{
  pid_t pid=(pid_t)0;
  if ((pid=fork()) == 0) {
    close(1);
    if (open("/tmp/Output",
        O_CREAT|O_WRONLY,
        0666) == -1) {
      perror("/tmp/Output");
      exit(1);
    }
    execl("/bin/date", "date", (char*)0);
    exit(1);
  }
  while(pid != wait(0)) ;
  return 0;
}
```



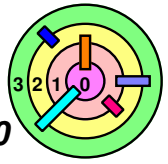
thr-term.c

```
#include "defs.h"
void *child(void *arg)
{
  if (*(int*)arg > 2) pthread_exit((void*)1);
  return((void*)2);
}
int main(int argc, char *argv[])
{
  pthread_t thread;
  void *result=NULL;
  pthread_create(&thread, 0, child, &argc);
  pthread_join(thread, (void**)&result);
  switch ((int)(long)result) {
  case 1: printf("result is 1\n"); break;
  case 2: printf("result is 2\n"); break;
  }
  return 0;
}
```



busywait.c

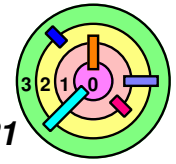
```
#include "defs.h"
#define NUM_THRS 100
int done[NUM_THRS];
pthread_t tid[NUM_THRS];
void *child(void *arg)
{
  int index=(int)(long)(arg);
  usleep(lrand48()%10000000);
  done[index] = 1;
  return 0;
}
int main(int argc, char *argv[])
{
  int i=0;
  memset(done, 0, sizeof(done));
  srand48(0);
  for (i=0; i < NUM_THRS; i++)
    pthread_create(&tid[i], 0, child, (void*)(long)i);
  waitall();
  return 0;
}
```



busywait.c

```
void waitall()
{
  for (;;) {
    int i=0, num_done=0;
    for (i=0; i < NUM_THRS; i++) {
      if (!done[i]) break;
      num_done++;
    }
    if (num_done == NUM_THRS) break;
  }
}
```

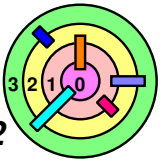
- Why is this busy wait?
  - the main thread is not doing anything useful
- Fix?
  - sleep for 100ms before checking
    - to avoid doing busy wait
    - not really a good solution



join.c

- Real fix
  - join with all threads

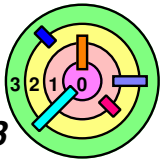
```
void waitall()
{
  int i=0;
  for (i=0; i < NUM_THRS; i++)
    pthread_join(tid[i], 0);
}
```



deadlock.c

Try to deadlock child thread and main thread

```
#include "defs.h"
void *child(void *arg)
{
  for (;;) { proc1(); }
  return((void*)0);
}

int main(int argc, char *argv[])
{
  pthread_t thread;
  srand48(time(0));
  pthread_create(&thread, 0, child, 0);
  for (;;) { proc2(); }
  return 0;
}
```

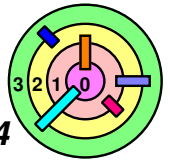


deadlock.c

```
pthread_mutex_t m1=PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t m2=PTHREAD_MUTEX_INITIALIZER;

void proc1() {
  pthread_mutex_lock(&m1);
  pthread_mutex_lock(&m2);
  printf("1");
  fflush(stdout);
  pthread_mutex_unlock(&m2);
  pthread_mutex_unlock(&m1);
  usleep(100000);
}
```

```
void proc2() {
  pthread_mutex_lock(&m2);
  pthread_mutex_lock(&m1);
  printf("2");
  fflush(stdout);
  pthread_mutex_unlock(&m1);
  pthread_mutex_unlock(&m2);
  usleep(100000);
}
```

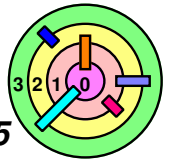
- Why no deadlock?
  - threads are alternating
- How to make it deadlock?
  - call `printf("-")` after locking `m1` in `proc1()` and call `printf("+")` after locking `m2` in `proc2()`
    - deadlock right away! (why?)



trylock.c

How to use `trylock()` to avoid deadlock

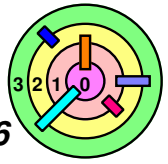
```
void proc2() {
  while (1) {
    pthread_mutex_lock(&m2);
    printf("+");
    fflush(stdout);
    if (!pthread_mutex_trylock(&m1))
      break;
    pthread_mutex_unlock(&m2);
  }
  printf("2");
  fflush(stdout);
  pthread_mutex_unlock(&m1);
  pthread_mutex_unlock(&m2);
  usleep(100000);
}
```



whoopee.c

- Let's catch SIGINT

```
#include "defs.h"
void handler(int signo)
{
  printf("Got signal %1d.  Whoopee!!\n", signo);
}
int main(int argc, char *argv[])
{
  sigset(SIGINT, handler);
  for (;;) { }
  return 1;
}
```



status-update.c

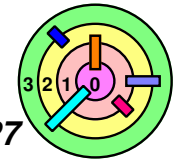
```
#include "defs.h"
typedef struct foo {
  int x, y;
} my_state;
my_state state;
int main() {
  state.x = state.y = 0;
  sigset(SIGINT, handler);
  long_running_proc();
  return 0;
}
```

```
void long_running_proc() {
  int i=0;
  for (i=0; i < 100; i++) {
    update_state(&state);
    compute_more();
  }
}
void handler(int signo) {
  display(&state);
}
```

```
void update_state(
    my_state *ptr) {
  ptr->x++;
  usleep(100000);
  ptr->y++;
}
```

```
void display(my_state *ptr) {
  printf("x = %1d\n", ptr->x);
  usleep(1000);
  printf("y = %1d\n", ptr->y);
}
```

```
void compute_more() { usleep(100000); }
```

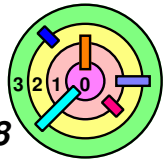


sigblock.c

- Let's block SIGINT

```
#include "defs.h"
typedef struct foo {
  int x, y;
} my_state;
my_state state;
sigset_t set;
int main() {
  state.x = state.y = 0;
  sigemptyset(&set);
  sigaddset(&set, SIGINT);
  sigset(SIGINT, handler);
  long_running_proc();
  return 0;
}
```

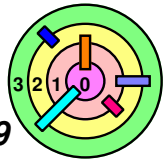
```
void long_running_proc() {
  int i=0;
  for (i=0; i < 100; i++) {
    sigset_t old_set;
    sigprocmask(SIG_BLOCK,
      &set, &old_set);
    update_state(&state);
    sigprocmask(SIG_SETMASK,
      &old_set, 0);
    compute_more();
  }
}
void handler(int signo) {
  display(&state);
}
```



sigwait.c

```
#include "defs.h"
typedef struct foo {
  int x, y;
} my_state;
my_state state;
sigset_t set;
pthread_mutex_t m=
PTHREAD_MUTEX_INITIALIZER;
int main() {
  pthread_t thr;
  state.x = state.y = 0;
  sigemptyset(&set);
  sigaddset(&set, SIGINT);
  sigprocmask(SIG_BLOCK,
    &set, 0);
  pthread_create(&thr, 0,
    monitor, 0);
  long_running_proc();
  return 0;
}
```

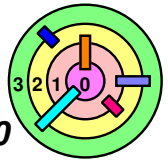
```
void long_running_proc() {
  int i=0;
  for (i=0; i < 100; i++) {
    pthread_mutex_lock(&m);
    update_state(&state);
    pthread_mutex_unlock(&m);
    compute_more();
  }
}
void *monitor(viod *arg) {
  int sig=0;
  for (;;) {
    sigwait(&set, &sig);
    pthread_mutex_lock(&m);
    display(&state);
    pthread_mutex_unlock(&m);
  }
  return 0;
}
```



direct.c

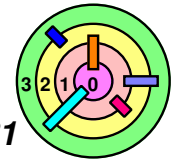
- Direct a thread to catch SIGINT
  - see "`man pthread_sigmask`" on Ubuntu 16.04

```
#include "defs.h"

#define handle_error_en(en, msg) \
  do { errno = en; perror(msg); exit(1); } while (0)

static void *
sig_thread(void *arg)
{
  sigset_t *set = arg;
  int s, sig;

  for (;;) {
    s = sigwait(set, &sig);
    if (s != 0)
      handle_error_en(s, "sigwait");
    printf("Signal handling thread got signal %d\n",
        sig);
  }
}
```



direct.c

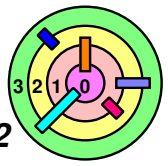
```
int
main(int argc, char *argv[])
{
  pthread_t thread;
  sigset_t set;
  int s;

  sigemptyset(&set);
  sigaddset(&set, SIGINT);
  sigaddset(&set, SIGUSR1);
  s = pthread_sigmask(SIG_BLOCK, &set, NULL);
  if (s != 0)
    handle_error_en(s, "pthread_sigmask");

  s = pthread_create(&thread, NULL, &sig_thread,
      (void *) &set);
  if (s != 0)
    handle_error_en(s, "pthread_create");

  pause();
}
```



cancel.c

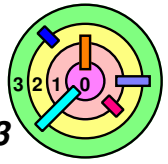
Cancellation

```
#include "defs.h"
#define NUM_THRS 10
pthread_t tid[NUM_THRS];
sigset_t set;
void cleanup(void *arg)
{
  int index=(int)(long)(arg);
  printf("Clean up thread %1d\n", index);
}
void *child(void *arg)
{
  pthread_cleanup_push(cleanup, arg);
  for (;;) {
    usleep(lrand48()%1000000);
  }
  pthread_cleanup_pop(0);
  return 0;
}
```



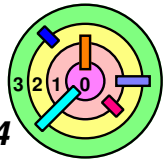
cancel.c

```
void waitall()
{
  int i=0;
  for (i=0; i < NUM_THRS; i++)
    pthread_join(tid[i], 0);
}
void *monitor(void *arg) {
  int i=0, sig=0;
  printf("Press <Ctrl+C>: ");
  fflush(stdout);
  sigwait(&set, &sig);
  printf("\nGot signal %1d\n", sig);
  for (i=0; i < NUM_THRS; i++) {
    while (tid[i] == 0) {
      usleep(100000);
    }
    pthread_cancel(tid[i]);
  }
  return 0;
}
```



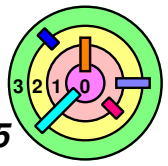
cancel.c

```
int main(int argc, char *argv[])
{
  pthread_t thr;
  int i=0;
  srand48(time(0));
  sigemptyset(&set);
  sigaddset(&set, SIGINT);
  sigprocmask(SIG_BLOCK, &set, 0);
  for (i=0; i < NUM_THRS; i++) {
    pthread_create(&tid[i], 0, child, (void*)(long)i);
  }
  pthread_create(&thr, 0, monitor, 0);
  waitall();
  pthread_join(thr, 0);
  return 0;
}
```



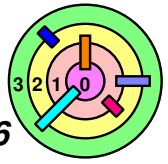
three-threads.c

```
/*
 * Demonstrate that only a thread with SIGINT unblocked
 *   will "see" SIGINT.
 * Currently, checking if num == 2 to unblock SIGINT
 *   for thread 2.  For all other threads, usleep() is
 *   never interrupted.
 *   Can change the comparison to 1 or 3 to demonstrate
 *     how to designate a different thread to only
 *     "see" SIGINT.
 */
#include <errno.h>
#include "defs.h"
sigset_t set;
void handler(int signo)
{
  printf("4");
  fflush(stdout);
}
```



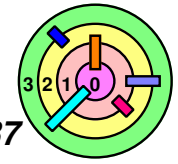
three-threads.c

```
void *child(void *arg) {
  int num=(int)(long)arg;
  if (num == 2) {
    sigprocmask(SIG_UNBLOCK, &set, 0);
  }
  for (;;) {
    printf("%1d", num);
    fflush(stdout);
    if (usleep(500000) != 0) {
      if (errno == EINTR) {
        printf("\nthread %1d interrupted...\n", num);
        fflush(stdout);
      }
    }
  }
  return 0;
}
```



three-threads.c

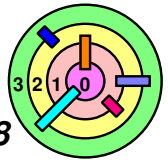
```
int main(int argc, char *argv[])
{
  int i=0;
  pthread_t t[3];
  sigset(SIGINT, handler);
  sigemptyset(&set);
  sigaddset(&set, SIGINT);
  sigprocmask(SIG_BLOCK, &set, 0);
  for (i=0; i < 3; i++)
    pthread_create(&t[i], 0, child, (void*)(long)(i+1));
  for (;;) {
    printf("0");
    fflush(stdout);
    usleep(500000);
    if (errno == EINTR) {
      printf("\nMain thread interrupted...\n");
      fflush(stdout);
    }
  }
  return 0;
}
```



Kernel Programming Assignments

Bill Cheng

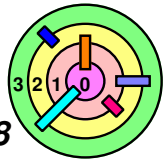
http://merlot.usc.edu/william/usc/



Kernel Programming Assignments (Part 1)

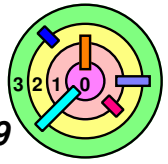
Bill Cheng

http://merlot.usc.edu/william/usc/



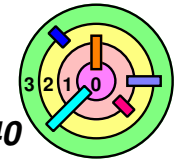
Academic Integrity Policy

- From the spec:
  - *accessing* a submission from a previous semester is considered cheating
    - e.g., you run a submission from a previous semester to see how another student's code works
  - if it's *evident* from your submission that you have *accessed* a submission from a previous semester, I will forward your case to USC Student Judicial Affairs for cheating investigation
  - the standard punishment is a grade of F in the class.
  - since there is only one submission from a team, *even if only one team member cheated*, *the entire team is considered cheated*.
  - therefore, please make sure that your teammates are not cheating!



Academic Integrity Policy

- Since the kernel assignments are group assignments, if *one of your teammate cheated*, the *entire group* is considered cheated
  - it is *your responsibility* to make sure that *your teammates are not cheating*
  - if you noticed that you teammate has written some code he/she cannot explain, you need to delete the code and write from scratch!
  - if you know that your teammate has code from previous semester, it's best you ask your teammate to *throw away the code*
- As mentioned in Lecture 1, posting `weenix` code (whether it's the original code or your code or a combination) to *public* `bitbucket` (or something similar) is considered *cheating*
  - posting warmup code to public `bitbucket` is also not permitted
  - you must *not* use `github.com`
  - can use `bitbucket.org`, but make sure it's private



Kernel Programming Assignments

- Tom Doeppner's *weenix* source and binary code
  - provided as `weenix-assignment-3.8.0.tar.gz`
  - incomplete
  - contains code like:

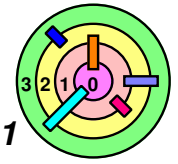
```
NOT_YET_IMPLEMENTED("PROCS: bootstrap");
```

assignment name

function name

    - your job is to implement these functions by *replacing* these lines with your code
      - please *replace* them *in-place*
      - do NOT *reorder* code there
  - to look for such code:

```
grep PROCS: kernel/*.c
grep PROCS: kernel/*/*.c
grep PROCS: kernel/*/*/*.c
grep PROCS: kernel/*/*/*/*.c
```



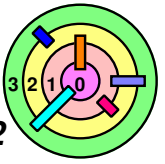
Download and Setup

```
% tar xvzf weenix-assignment-3.8.0.tar.gz
% cd weenix-assignment-3.8.0/weenix
% make clean
% make
% ./weenix -n
```

- Don't type the long commands, just copy from your web browser and paste into your terminal
  - if something looks wrong, seek help as soon as you can
  - best *not* to do the above in a Shared Folder
    - see kernel FAQ if you have to do that

- Don't forget to install these when you setup Ubuntu 16.04:

```
sudo apt-get install -y git
sudo apt-get install -y gitk
sudo apt-get install -y git-core
sudo apt-get install -y qemu
sudo apt-get install -y cscope
```



Download and Setup

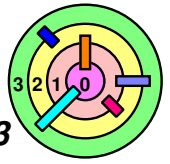
- If all goes well, you should see tons of stuff fly by and the following at the bottom of the terminal:

```
Not yet implemented: PROCS: bootstrap, file
    main/kmain.c, line 184
panic in main/kmain.c:186 bootstrap():
    weenix returned to bootstrap()!!! BAD!!!

Kernel Halting.
```

  - press <Ctrl+C> in the terminal to quit

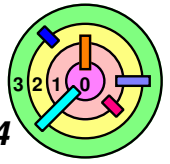
- Make sure you have tried the above this weekend
  - any problem, let us know *NOW*

- Don't worry that you don't know how to do kernel 1
  - for now, just get familiar with the spec and the documentation
    - and may be read some kernel code
    - you are *not* expected to understand a lot of the kernel code



Documentation

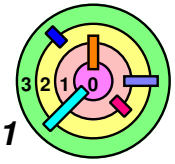
- The `weenix` documentation is in `doc/latex/documentation.pdf`
  - introduces `weenix` to you
  - detailed instructions on assignments
  - you must read it thoroughly
- We are doing three of the assignments
  - *Processes and Threads (PROCS)*
  - *Virtual File System (VFS)*
  - *Virtual Memory (VM)*
- We are *not* doing two of the assignments
  - Drivers (DRIVERS)
  - System V File System (S5FS)
  - these are done for you and they are compiled and provided as libraries
    - `kernel/libdrivers.a` and `kernel/libs5fs.a`
      - source code for these are *not* available; you need to learn how to *work around* code you don't have



Kernel Programming Assignments (Part 2)

Bill Cheng

http://merlot.usc.edu/william/usc/



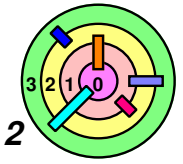
Compilation and Configuration

- `Config.mk` controls what gets compiles and configured into the kernel (weenix is a monolithic OS)
  - for PROCS, use the original `Config.mk`
    - set `DRIVERS` to 1 to complete this assignment
  - for VFS, set `DRIVERS` and `VFS` to 1
  - for VM, set `DRIVERS`, `VFS`, `S5FS`, and `VM` to 1
    - set `VM` to 0 at first to get `kernel/mm/pframe.c` working
    - then set `VM` to 1 to work on the rest of the assignment
    - set `DYNAMIC` to 1 in the end if everything is working
  - by default: `DBG = all`
    - the grader will use these for ***grading:***

```
DBG = error,test
DBG = error,print,test
```

- Every time you modify `Config.mk`, you should do:

```
% make clean
% make
% ./weenix -n
```



Debugging with gdb

- The `weenix` documentation says to do this to debug the kernal:

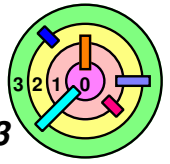
```
% ./weenix -n -d gdb
```

  - although you can use `gdb`, but you cannot see kernel debugging messages (from `dbg()` calls)

- To see kernel debugging messages AND debug the kernel, do:
  - set `GDBWAIT=1` in `Config.mk` then *recompile* kernel
  - run `weenix` under gdb with:

```
% ./weenix -n -d gdb -w 10
```

    - if you have a slow machine, you should use a larger value
    - if you have a fast machine, you should use a smaller value
  - unfortunately, if you have compiled with `GDBWAIT=1` and want to run *without gdb*, `weenix` will *freeze*
    - you have to set `GDBWAIT` back to `0` and recompile if you want to run `weenix` without gdb
    - you should set `GDBWAIT` back to `0` when you submit your assignment for grading

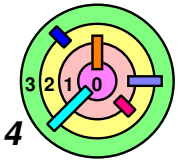


Submissions

- Processes and Threads (PROCS)

```
% make procs-submit
tar cvzf procs-submit.tar.gz \
        Config.mk \
        procs-README.txt \
        ...
```

  - must fill out `procs-README.txt`, it's your assignment's documentation
    - *IMPORTANT:* your must copy the `k1-README.txt` template in the spec into `procs-README.txt` and make changes
    - this is where you should also include
      1) how to split the points (in terms of percentages and must sum to 100%)
      2) brief justification about the split (if not equal split)
         - please understand that if I have to get involved, it can also be unfair since I won't have all the information
         - best to claim even splits
  - submit `procs-submit.tar.gz` using web form



Submissions

- Virtual File System (VFS)
  - need to fill out `vfs-README.txt` (start with `k2-README.txt`)
    - start with the `k2-README.txt` template in the spec

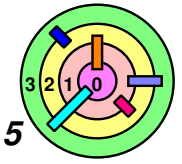
      ```
      % make vfs-submit
      ```

- Virtual Memory (VM)
  - need to fill out `vm-README.txt` (start with `k3-README.txt`)
    - start with the `k3-README.txt` template in the spec

      ```
      % make vm-submit
      ```

- Must *NOT* include *ANY OTHER* file not mentioned in "`make`" above
  - or we will *delete* it before grading
  - you must not add any files

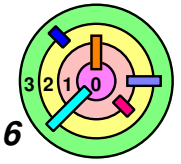
- Submit source code only
  - we will deduct 2 points if you submit binary files
  - we will deduct 2 points if you submit extra files
  - we will deduct 2 points if you do not keep the same directory structure



Verify Your Kernel Submission

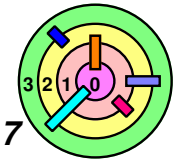
- Assume that in your home directory, you have
  - a *pristine* `weenix-assignment-3.8.0.tar.gz`
  - your submission file, e.g., `procs-submit.tar.gz`
- Do the following to verify your submission

```
% rm -rf /tmp/xyzzy
% mkdir /tmp/xyzzy
% cd /tmp/xyzzy
% tar xvzf ~/weenix-assignment-3.8.0.tar.gz
% cd weenix-assignment-3.8.0/weenix
% tar xvzf ~/procs-submit.tar.gz
% make clean
% make
% ./weenix -n
[ go through grading guidelines line by line ]
[ check every line of your README file ]
```



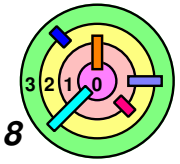
Grading

- The weenix code and comments and the weenix documentation are from Brown University
  - these are just hints and not our "requirements"
- Just like the warmup assignments, the *spec* and the *grading guidelines* are the requirements for our CS 402
  - they are related to "grading"
- Read the grading gudelines carefully!
  - e.g., altering or removing top comment block in a submitted `.c` file will *cost you 20 points* each!
    - why such stiff penalty?
      - because it's extremely easy to follow this rule
  - e.g., altering/removing a call to `dbgq()` in `bootstrap()` in your submitted `kmain.c` file will *cost you 20 points*
    - this is a signature showing who downloaded the kernel source
  - designate a teammate to check your submission against *every line in the grading guidelines*



Structure Of Grading Guidelines

- "Plus Points" section of the grading guidelines
  - *mandatory KASSERTs:* section (A)
    - we are actually trying to help you to write some of your kernel code and give you some easy points at the same time!
    - add `KASSERT()` calls (followed by a "conforming `dbg()` call")
  - *SELF-checks:* last section
    - *every code sequence* inside any function you wrote to replace a `NOT_YET_IMPLEMENTED()` call must *END* with a "conforming `dbg()` call"
      - must use *"static analysis"* (independent of how your code will actually execute) to analyze your code
  - *PRE-CANNED tests:* other sections
- Please read the grading guidelines very carefully
  - when in doubt, ask the instructor!

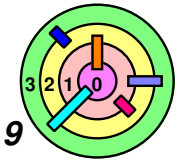


"Conforming `dbg()` Call"

- "Conforming `dbg()` call"
  - general form:

```
dbg(DBG_PRINT, "(GRADING#S X.Y)\n");
```

  - `#` is the kernel assignment number, can only be `1`, `2`, or `3`
  - `S` is the section number of the grading guidelines
    - for section (A) of the grading guidelines, you must use the corresponding `X` and `Y`
      - read the requirements very very carefully!
    - for other sections of the grading guidelines:
      - `X` is a subtest number (applicable only when the subtest can be run separately with a shell-level command)
      - never use `X.Y`
  - you must *format it exactly* according to spec or you will lose a lot of points
    - read the requirements very very carefully!
  - when in doubt, ask the instructor!



"Conforming `dbg()` Call"

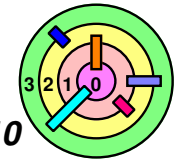
- There are only *two* reasons to make a "conforming `dbg()` call"
  1) get credit for an item in *section (A)* of the grading guidelines
     - must use this form (must specify `#`, `X`, and `Y`):

```
dbg(DBG_PRINT, "(GRADING#A X.Y)\n");
```

  2) get credit for *SELF-checks*
     - must use this form (use `X` if applicable):

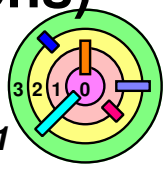
```
dbg(DBG_PRINT, "(GRADING#A)\n");
dbg(DBG_PRINT, "(GRADING#B X)\n");
dbg(DBG_PRINT, "(GRADING#C X)\n");
...
```

     - `dbg(DBG_PRINT, "(GRADING#A)\n");` means start and stop the kernel (without doing anything else)
- when in doubt, more is better than less!



"SELF-checks"

- As part of our requirements, you must ***not*** put/leave useless stuff in your kernel (i.e., don't leave trash in the kernel)
  - ***every code sequence must be traversable***
    - a code sequence is a block of code that does not contain conditionals or gotos and has only one entry point at the top
      - a code path is a path your code ***may*** take, i.e., there is a way to execute your code along that path
    - if a code sequence is not traversable, you must ***delete*** it
      - ***must not leave useless code in the kernel!***
    - if you cannot demonstrate that there is way to get to it, you must ***remove*** it or we will take points off
- This is referred to as "SELF-checks" in the spec
  - when you are confused about "SELF-checks", please come back and read this slide again to remind yourself why we are doing "SELF-checks" (and hopefully, that will answer your questions)



"SELF-checks"

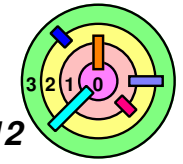
- What's useless code?

```
if (cond) {
  /* code seq */
}
```

  - if `cond` can never be true, then `code seq` is *useless*
  - to demonstrate that `code seq` is useful, you must tell the grader *which test to run* to reach the *END* of `code seq`
    - must add "conforming `dbg()` call" at the *END* of `code seq`

```
if (cond) {
  /* code seq */
  dbg(DBG_PRINT, "(GRADING#S X)\n");
}
```

      - `#` is assignment number: `1`, `2`, or `3`
      - `S` is section number: `B`, `C`, `D`, ...
      - `X` is subsection number (if applicable)
    - just need to tell the grader *one way* to get there



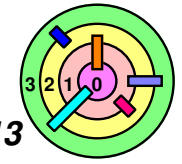
Must Use Static Analysis For SELF-checks

- To determine where you must add "conforming dbg() call", you must perform *"static analysis"* of your code (i.e., does not depend on how your code actually runs)

```
while (cond) {
  /* seq1 */
}
/* seq2 */
```

  - you may argue that the first time the while loop is executed, `cond` will be true, then later on, `cond` will be false, so you just need a "conforming dbg() call" at the end of `seq2`
    - that would be an incorrect analysis because it depends on how your code would execute
    - in this case, you must put a "conforming dbg() call" at the END of `seq1` and at the END of `seq2`

- Please read the spec for details!



"Conforming `dbg()` Call" Requirement

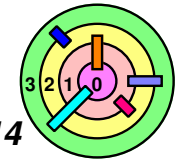
Example from spec:

```
/* sequence1 */
if (cond1) {
   /* sequence2 */
} else {
   /* sequence3 */
}
/* sequence4 */
```

```
/* sequence1 */
if (cond1) {
   /* sequence2 */
   /* conforming dbg() call */
} else {
   /* sequence3 */
   /* conforming dbg() call */
}
/* sequence4 */
/* conforming dbg() call */
```

A "sequence" is a list of C statements *not* containing conditionals, gotos, or a label which is the target of a goto statement

- if you are not sure, you can make a "conforming `dbg()` call" at the *END* of *every code sequence*
  - you cannot go wrong with more!



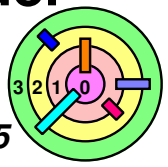
Backing Up Your Work & Collaboration

- You *have to* have a plan to *backup your code* and backup routinely
  - if you lose your work, no one can recover your files
  - can use *private* DropBox / iCloud / Microsoft cloud
  - or use a *private* `bitbucket` (must *not* use `github`)
    - share it among your team members only

- One simple way to backup your work
  - at the end of each day, do:

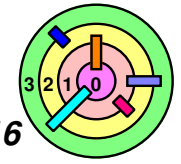
```
% make backup
```

    - it will tell you the name of the backup file
    - if you have a "`Shared-ubuntu`" shared folder in your home directory, your backup file will be copied there
      - if not, your backup file will be in the current directory and you should copy it into a shared folder
  - use your host's cloud backup facility to back up this file/folder



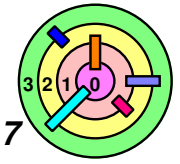
Backing Up Your Work & Collaboration

- You should use `git` to collaborate among project partners
  - read "Pro Git" (a free online book, one of our textbooks)
- But you need a shared repository in the cloud to collaborate with your teammates
  - there are free `git` repositories on the web
    - unfortunately, most of the free ones are required to be visible by the world - you must *not* use these
      - on `github.com`, private repository automatically becomes public after 2 years (if you don't pay)
      - this is why you must *not* use `github.com`
    - you can use `bitbucket.org` but you need to make sure that your files remain private
  - apply for *free academic account* on `bitbucket.org` to share with teammates
    - make sure your projects are truly *private*



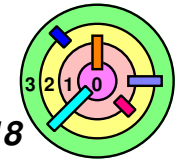
Backing Up Your Work & Collaboration

- If you have two people working on the same file and then update the repository one after another
  - `git` will attempt to merge the changes, but it may not be what you want
  - may be it's best to *coordinate* and not have two pepole modifying the same file



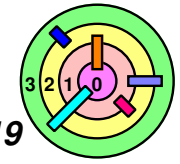
Early and Late Policies

- Same early submission policy as warmup projects
- Similar late submission policy as warmup projects
  - except that kernel 1 can be submitted by the kernel 2 deadline and get a 50% deduction
  - similarly, kernel 2 can be submitted by the kernel 3 deadline and get a 50% deduction
  - kernel 3 has regular late submission policy



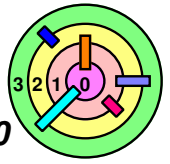
Extra Credit

- You can get *extra credit* for posting *timely* and *good/useful* answers to the class Google Group in response to questions posted by other students regarding *kernel* programming assignments
  - the maximum number of extra credit points you can get is *10* points for each of the kernel assigments (on a 100-point scale)
  - "timely" means *within 8 hours of the original post*
    - if you don't *"reply"* to another student's post, you are not eligible for this type of extra credit
  - if you just repeat what others are saying, you are being "helpful" but what you post will not be "useful" since it's already been said



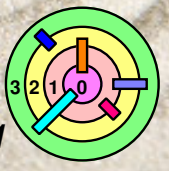
How Do You Start?

- Definitely start with the documentation, spec, and kernel FAQ
- Read code, read lots and lots of code
  - try things out and see what happens (debugging statements)
  - you need to *absorb* other people's code, make sense of it
    - although you *don't have to understand everything*
  - that's what OS hacking (in the good sense) is all about
    - it's *not* about "implementing an OS"
- It's the *process* that matters
  - *not* the *answers*
  - it's about learning how to figure out *which* 2 or 3 lines (or 20 or 30 lines) of code to insert and *where*
- So, it needs to be experienced
  - you should not expect quick/straight answers
  - this is not an OS hacking class
- Learning to write OS code is like...



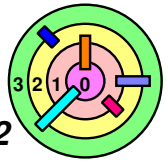
How Do You Start?

- Definitely start with the documentation, spec, and kernel FAQ
- Read code, read lots and lots of code
  - try things out and see what happens (debugging statements)
  - you need to *absorb* other people's code, make sense of it
    - although you *don't have to understand everything*
  - that's what OS hacking (in the good sense) is all about
    - it's *not* about "implementing an OS"
- It's the *process* that matters
  - *not* the *answers*
  - it's about learning how to figure out *which* 2 or 3 lines (or 20 or 30 lines) of code to insert and *where*
- So, it needs to be experienced
  - you should not expect quick/straight answers
  - this is not an OS hacking class
- Learning to write OS code is like... *Zen*



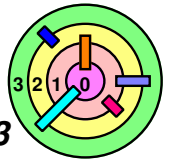
Getting Help

- If you have questions about the kernel assignments
  - read documentation, textbook, lecture slides, read more code
  - send me e-mail
    - please understand that neither I nor other teaching staff can *tell you what code to write*!
    - if you ask me if you should set this variable to x or y, I will ask you to try both and figure out what makes more sense!
      - if you send us questions like that, we may simply forward your post to the class Google Group since we cannot tell you what code to write
  - post your questions to the class Google Group
    - your classmates are a great resource!
    - sometimes, we may not immediately answer these questions to give your classmates an opportunity to earn extra credit points
      - we may wait 2 hours



Pitfalls To Avoid

- Your team need to *meet often*
  - once a day is preferred
    - work at the same place at the same time
    - have lots of discussions (and write a fair amount of code)
  - swallow your pride, be honest with your teammates, don't hide your weakness
    - everyone gets the same grade
    - if no one is really good at this (which is not unusual), someone (or more) has to step up
- You don't have to know what every piece of code is doing
  - learn how to *assume* that other code works (until proven otherwise)
    - other code works kind of like what's covered in lectures
  - use "`grep`" to *get an idea* of how a function is used and how a field in a data structure is used

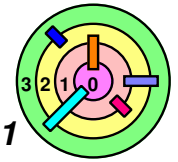


Kernel 1

Bill Cheng

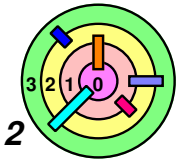
http://merlot.usc.edu/william/usc/

- **Prerequisite: a simple system (Ch 4)**
  - **when you finish the prerequisite, please come back and review this material**



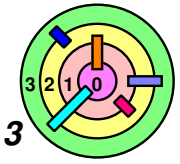
Boot Prestine Kernel

```
% script
Script started, file is typescript
reading .login (xterm) ...
% ./weenix -n
/usr/bin/qemu-system-i386
...
Not yet implemented: PROCS: bootstrap, file \
main/kmain.c, line 184
panic in main/kmain.c:186 bootstrap(): weenix \
returned to bootstrap()!!! BAD!!!

Kernel Halting.

^C
qemu: terminating on signal 2
% exit
Script done, file is typescript
% more typescript
```



Look At "typescript"

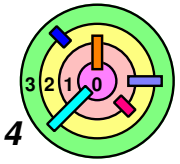
- Where is kernel's text segment?
  - `0xc0000000-0xc0038000` means `[0xc0000000,0xc0038000)`
    - this interval is "closed" on the left (i.e., includes) and "open" on the right (i.e., excludes)
      - pretty much all intervals are denoted this way
- Where are the other segments?
  - data: `[0xc0038000-0xc0044baa)`
  - bss: `[0xc0044baa-0xc005a000)`
  - page system: `[0xc00a0000-0xcfe9d000)`
    - I think this is like the dynamic segment for the kernel (i.e., Buddy System)
- If your numbers are slightly different, it's probably okay



Look At "typescript"

- What else?
  - kernel data structures are initialized (e.g., memory allocators)
  - some hardware are initialized (e.g., PIC)
    - *don't need to understand EVERYTHING*
  - if your kernel access anywhere outside this range, most likely you will get a *kernel page fault*
    - this will be followed by a *kernel panic*

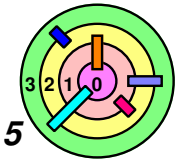
- *VERY IMPORTANT:* read kernel FAQ about *"how can I debug a page fault that caused kernel panic?"*
  - whenever you get a kernel page fault, you must follow the steps there to figure out *EXACTLY* where your kernel crashed
    - I have no special power, I have no idea why your kernel crashed



Kernel 1

```
% make nyi
```

proc/kmutex.c:36	kmutex_init()	PROCS
proc/kmutex.c:48	kmutex_lock()	PROCS
proc/kmutex.c:58	kmutex_lock_cancellable()	PROCS
proc/kmutex.c:78	kmutex_unlock()	PROCS
proc/kthread.c:106	kthread_create()	PROCS
proc/kthread.c:127	kthread_cancel()	PROCS
proc/kthread.c:148	kthread_exit()	PROCS
proc/proc.c:223	proc_create()	PROCS
proc/proc.c:254	proc_cleanup()	PROCS
proc/proc.c:268	proc_kill()	PROCS
proc/proc.c:280	proc_kill_all()	PROCS
proc/proc.c:294	proc_thread_exited()	PROCS
proc/proc.c:315	do_waitpid()	PROCS
proc/proc.c:328	do_exit()	PROCS
proc/sched.c:121	sched_cancellable_sleep_on()	PROCS
proc/sched.c:137	sched_cancel()	PROCS
proc/sched.c:179	sched_switch()	PROCS
proc/sched.c:198	sched_make_runnable()	PROCS
proc/sched_helper.c:43	sched_sleep_on()	PROCS
proc/sched_helper.c:49	sched_wakeup_on()	PROCS
proc/sched_helper.c:56	sched_broadcast_on()	PROCS
main/kmain.c:184	bootstrap()	PROCS
main/kmain.c:279	initproc_create()	PROCS
main/kmain.c:298	initproc_run()	PROCS



Kernel 1

Kernel thread creation, cancellation, and destruction

proc/kthread.c:106	kthread_create()	PROCS
proc/kthread.c:127	kthread_cancel()	PROCS
proc/kthread.c:148	kthread_exit()	PROCS

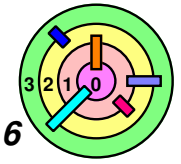
- please keep `MPT=0` in `Config.mk` and *only* implement *single thread processes*

Kernel scheduler

proc/sched.c:121	sched_cancellable_sleep_on()	PROCS
proc/sched.c:137	sched_cancel()	PROCS
proc/sched.c:179	sched_switch()	PROCS
proc/sched.c:198	sched_make_runnable()	PROCS
proc/sched_helper.c:43	sched_sleep_on()	PROCS
proc/sched_helper.c:49	sched_wakeup_on()	PROCS
proc/sched_helper.c:56	sched_broadcast_on()	PROCS

Kernel process creation, waiting, and destruction

proc/proc.c:223	proc_create()	PROCS
proc/proc.c:254	proc_cleanup()	PROCS
proc/proc.c:268	proc_kill()	PROCS
proc/proc.c:280	proc_kill_all()	PROCS
proc/proc.c:294	proc_thread_exited()	PROCS
proc/proc.c:315	do_waitpid()	PROCS
proc/proc.c:328	do_exit()	PROCS



Kernel 1

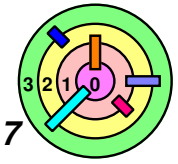
Kernel mutex

proc/kmutex.c:36	kmutex_init()	PROCS
proc/kmutex.c:48	kmutex_lock()	PROCS
proc/kmutex.c:58	kmutex_lock_cancellable()	PROCS
proc/kmutex.c:78	kmutex_unlock()	PROCS

Kernel startup

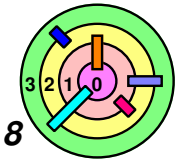
main/kmain.c:184	bootstrap()	PROCS
main/kmain.c:279	initproc_create()	PROCS
main/kmain.c:298	initproc_run()	PROCS

- Read the *comment blocks* to figure out what these functions suppose to do and how they are related to each other
  - use "`grep`" to see how they are called
  - feel free to discuss this in the class Google Group
  - you can talk about code that came with the prestine kernel
    - do *not* post code/pseudo-code you are planning to write
- When a thread gives up the CPU, you must make sure that all *global variables* and *data structures* are in a *consisten state*



How To Start?

- I would recommend doing this first:
  - phase 1: get the kernel to simply start and quit with `DRIVERS=0`
    - in `bootstrap()`, create IDLE proc and switch to IDLE proc
      - first procedure of IDLE proc is written for you already, don't change anything
      - *Hint:* look at the code in `faber_thread_test()` to see how to create a process, create a thread in it, run it
      - Note: code in `faber_thread_test()` is running in *thread context*, you are *not* in a thread context in `bootstrap()`
    - in `initproc_create()`, create INIT proc and INIT thread
    - don't write code in `initproc_run()`, which is the first procedure of INIT proc
      - it should self-terminate
      - single-step to see where it goes
    - this should wake up the IDLE proc, which will turn off the machine

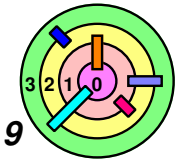


How To Start?

- I would recommend doing this first:
  - phase 1: get the kernel to simply start and quit with `DRIVERS=0`
    - make sure the *kernel halts cleanly*
    - these functions are involved (the list may *not* be complete):

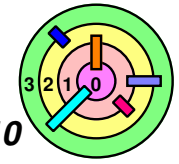
```
proc/kthread.c:106          kthread_create()     PROCS
proc/kthread.c:148            kthread_exit()     PROCS
   proc/proc.c:223             proc_create()     PROCS
   proc/proc.c:254            proc_cleanup()     PROCS
   proc/proc.c:294       proc_thread_exited()    PROCS
   proc/proc.c:315             do_waitpid()     PROCS
  proc/sched.c:121  sched_cancellable_sleep_on()  PROCS
  proc/sched.c:179           sched_switch()     PROCS
  proc/sched.c:198     sched_make_runnable()    PROCS
proc/sched_helper.c:43       sched_sleep_on()   PROCS
proc/sched_helper.c:49      sched_wakeup_on()   PROCS
proc/sched_helper.c:56    sched_broadcast_on()  PROCS
      main/kmain.c:184            bootstrap()   PROCS
      main/kmain.c:279        initproc_create() PROCS
```

    - yes, it's a lot of code to get working just for phase 1!
    - in a way, phase 1 is the most important step
      - if you do it the wrong way, you will have to come back and fix your code



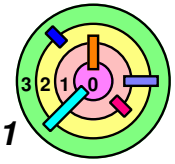
How To Start?

- I would recommend doing this first:
  - phase 2: get the kernel to simply start and quit *cleanly*
    - call `faber_thread_test()` from `initproc_run()`
      - start with `CS402TESTS=1` in `Config.mk`
      - then set `CS402TESTS=2`, `3`, and so on, up to `8`
    - always make sure the *kernel halts cleanly*
  - phase 3: set `DRIVERS=1` in `Config.mk`
    - run `kshell` in `initproc_run()`
      - "`help`", "`echo`" and "`exit`" `kshell` commands should work
    - add `kshell` commands to invoke any test function in grading guidelines and your README (see rules about "SELF-checks")
      - for each `kshell` command, you need to *create a child process* and set the test function as the *first procedure* of the thread in the child process



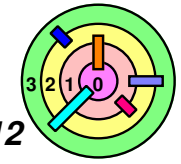
How To Start?

- Please remember that the teaching staff *cannot tell you what code to write*
- Here's what's appropriate to talk about in class Google Group
  1) the spec
  2) the kernel FAQ
  3) the grading guidelines
  4) test code
     - kernel 1: `faber_thread_test()`, `sunghan_test()`, `sunghan_deadlock_test()`
       - they are mentioned in the grading guidelines
       - your kernel needs to work with these test code perfectly



Hints?

- Hints are all over the place!
  - documentation
  - spec
  - comment in code
  - kernel code itself
  - kernel FAQ
- For example, the spec says:
  - the kernel is very very powerful
    - if there is a bug, it's *your* bug!
  - the weenix kernel is *non-preemptive*
    - *non-preemptive* means that a thread *cannot be preempted* by *another thread*
      - it can be interrupted to *service an interrupt*, then goes back to what it was doing before
      - if a kernel thread *does not want to be cancelled*, there is *no way* to kill it
  - we are *not* implementing multiple threads per process, i.e., `MTP=0` in `Config.mk`



"kmain.c"

kmain()

```
context_setup(&bootstrap_context, bootstrap, 0, NULL, bstack,
              PAGE_SIZE, bpdir);
context_make_active(&bootstrap_context);
panic("\nReturned to kmain()!!!\n");
```

bootstrap()

```
NOT_YET_IMPLEMENTED("PROCS: bootstrap");
panic("weenix returned to bootstrap()!!! BAD!!!\n");
```

idleproc_run()

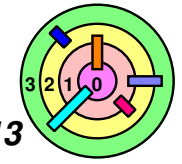
```
kthread_t *initthr = initproc_create();
init_call_all();
GDB_CALL_HOOK(initialized);
intr_enable();
sched_make_runnable(initthr);
child = do_waitpid(-1, 0, &status);
```

initproc_create()

```
NOT_YET_IMPLEMENTED("PROCS: initproc_create");
```

initproc_run()

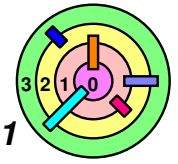
```
NOT_YET_IMPLEMENTED("PROCS: initproc_run");
```



More On Kernel 1

Bill Cheng

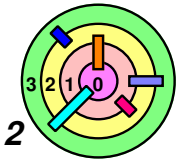
http://merlot.usc.edu/william/usc/



Kernel Code

- We will go over *some* kernel code now (we will not put all the code in "`faber_test.c`" and "`sunghan_test.c`" on these slides)
  - we will not cover all the test code
    - we will probably cover a *small number of test cases*
    - you need to learn how to read all these code and use them to figure out what you need to do
  - if you have a specific test you want the instructor to talk about next Friday, please send him an e-mail before 5pm on Thursday

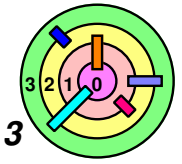
- *IMPORTANT:* at any line in `faber_thread_test()`, ask yourself
  1) where are all the threads/processes?
     - i.e., in which *queue* is a thread sleeping
  2) if a thread is not in the run queue, *who* is going to *unblock* it and *when/how* (by calling what function)?
  3) exactly how and where will a particular process/thread die?



faber_thread_test()

- Let's look at the first set of tests in `faber_thread_test()`
  - first subtest

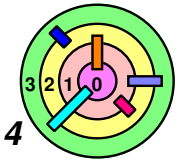
```
dbg(DBG_TEST, "waitpid any test\n");
start_proc(&pt, "waitpid any test", waitpid_test, 23);
wait_for_any();

/*
 * Create a process and a thread with the given name
 *   and calling teh given function. Arg1 is passed to
 *   the function (arg2 is always NULL).  The thread
 *   is immediately placed on the run queue.  A
 *   proc_thread_t is returned, giving the caller a
 *   pointer to the new process and thread to
 *   coordinate tests.  NB, the proc_thread_t is
 *   returned by value, so there are no stack problems.
 */
static void start_proc(proc_thread_t *ppt, char *name,
    kthread_func_t f, int arg1) {
  ...
}
```



faber_thread_test()

- Let's look at the first set of tests in `faber_thread_test()`
  - first subtest

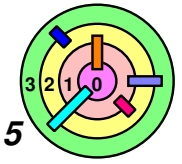
```
static void start_proc(proc_thread_t *ppt, char *name,
    kthread_func_t f, int arg1) {
  proc_thread_t pt;

  pt.p = proc_create(name);
  pt.t = kthread_create(pt.p, f, arg1, NULL);
  KASSERT(pt.p && pt.t &&
      "Cannot create thread or process");
  sched_make_runnable(pt.t)
  if (ppt != NULL) {
    memcpy(ppt, &pt, sizeof(proc_thread_t));
  }
}

void *waitpid_test(int arg1, void *arg2) {
  do_exit(arg1);
  return NULL;
}
```



faber_thread_test()

- Let's look at the first set of tests in `faber_thread_test()`
  - first subtest

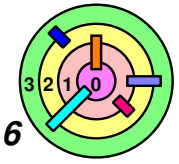
```
/**
 * Call waitpid with a -1 pid and print a message
 *   about any process that exits.
 * Returns the pid found, including -ECHILD when this
 *   process has no children.
 */
static pid_t wait_for_any() {
  int rv;
  pid_t pid = do_waitpid(-1, 0, &rv);
  if (pid != -ECHILD)
    dbg(DBG_TEST, "child (%d) exited: %d\n", pid, rv);
  return pid;
}
```



faber_thread_test()

- Let's look at the first set of tests in `faber_thread_test()`
  - 2nd subtest

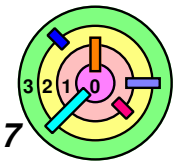
```
dbg(DBG_TEST, "waitpid test\n");
start_proc(&pt, "waitpid test", waitpid_test, 32);
pid = do_waitpid(2323, 0, &rv);
if ( pid != -ECHILD )
  dbg(DBG_TEST, "Allowed wait on non-existent pid\n");
wait_for_proc(pt.p);

void *waitpid_test(int arg1, void *arg2) {
  do_exit(arg1);
  return NULL;
}
```



faber_thread_test()

- Let's look at the first set of tests in `faber_thread_test()`
  - 2nd subtest

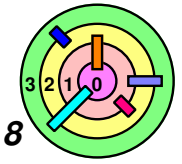
```
/**
 * Call do_waitpid with the process ID of the given
 *   process.  Print a debug message with the exiting
 *   process's status.
 */
static void wait_for_proc(proc_t *p) {
  int rv;
  pid_t pid;
  char pname[PROC_NAME_LEN];

  strncpy(pname, p->p_comm, PROC_NAME_LEN);
  pname[PROC_NAME_LEN-1] = '\0';
  pid = do_waitpid(p->p_pid, 0, &rv);
  dbg(DBG_TEST, "%s (%d) exited: %d\n", pname, pid, rv);
}
```



faber_thread_test()

- Let's look at the first set of tests in `faber_thread_test()`
  - 3rd subtest

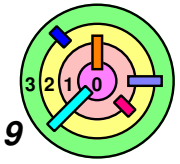
```
dbg(DBG_TEST, "kthread exit test\n");
start_proc(&pt, "kthread exit test",
    kthread_exit_test, 0);
wait_for_proc(pt.p);

/*
 * A thread function that returns NULL, silently
 *   invoking kthread_exit()
 */
void *kthread_exit_test(int arg1, void *arg2) {
    return NULL;
}
```



faber_thread_test()

- Let's look at the first set of tests in `faber_thread_test()`
  - 4th subtest

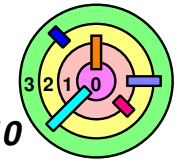
```
dbg(DBG_TEST, "many test\n");
for (i = 0; i < 10; i++)
  start_proc(NULL, "many test", waitpid_test, i);
wait_for_all();
dbg(DBG_TEST, "(C.1) done\n");


void *waitpid_test(int arg1, void *arg2) {
  do_exit(arg1);
  return NULL;
}

/* Repeatedly call wait_for_any() until it returns
 *   -ECHILD */
static void wait_for_all() {
    while (wait_for_any() != -ECHILD)
        ;
}
```



faber_thread_test()

- Let's look at the first set of tests in `faber_thread_test()`
  - 4th subtest

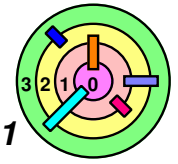
```
/**
 * Call waitpid with a -1 pid and print a message
 *   about any process that exits.
 * Returns the pid found, including -ECHILD when this
 *   process has no children.
 */
static pid_t wait_for_any() {
  int rv;
  pid_t pid = do_waitpid(-1, 0, &rv);
  if (pid != -ECHILD)
    dbg(DBG_TEST, "child (%d) exited: %d\n", pid, rv);
  return pid;
}
```



More Test Code

- We will not put all the code in "`faber_test.c`" and "`sunghan_test.c`" on these slides
  - in the discussion sections, we will discuss more of these test code
    - we will not cover all the test code
      - you need to learn how to read all these code and use them to figure out what you need to do
      - we will probably cover a *small number of test cases*
  - if you have a specific test you want the TA to talk about, please send him an e-mail before 5pm on Thursday

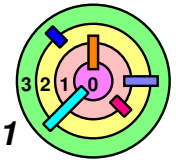
- *IMPORTANT:* at any line in `faber_thread_test()`, ask yourself
  1) where are all the threads/processes?
     - i.e., in which *queue* is a thread sleeping
  2) if a thread is not in the run queue, *who* is going to *unblock* it and *when/how* (by calling what function)?



Kernel 2

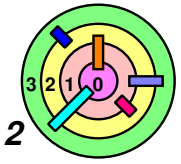
Bill Cheng

http://merlot.usc.edu/william/usc/



New Things in Kernel 2

- Polymorphism - VFS should be able to work with *any* AFS
  - achieved using *polymorphism*
    - the AFS for kernel 2 is `ramfs`
    - read the kernel code to see how it works
- Reference counting
  - whenever you *keep* a reference to an object, you increase the reference count of that object
    - so that the object won't disappear
  - whenever you remove a reference to an object, you decrement the reference count of that object
    - if it reached zero, it means that nothing in the system knows about that object and you should delete/free it
  - this is done for objects in memory
  - this is done for objects inside the file system (which you don't have to worry about)

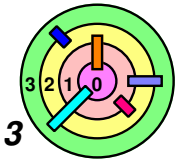


Kernel 2

```
% make nyi
```

```
        main/kmain.c:218                    idleproc_run()      VFS
        main/kmain.c:223                    idleproc_run()      VFS
          fs/vnode.c:460                special_file_read()     VFS
          fs/vnode.c:473               special_file_write()     VFS
       fs/vfs_syscall.c:68                       do_read()      VFS
       fs/vfs_syscall.c:83                      do_write()      VFS
       fs/vfs_syscall.c:97                      do_close()      VFS
      fs/vfs_syscall.c:120                        do_dup()      VFS
      fs/vfs_syscall.c:136                       do_dup2()      VFS
      fs/vfs_syscall.c:168                      do_mknod()      VFS
      fs/vfs_syscall.c:189                      do_mkdir()      VFS
      fs/vfs_syscall.c:214                      do_rmdir()      VFS
      fs/vfs_syscall.c:235                     do_unlink()      VFS
      fs/vfs_syscall.c:263                       do_link()      VFS
      fs/vfs_syscall.c:278                     do_rename()      VFS
      fs/vfs_syscall.c:298                      do_chdir()      VFS
      fs/vfs_syscall.c:320                    do_getdent()      VFS
      fs/vfs_syscall.c:337                      do_lseek()      VFS
      fs/vfs_syscall.c:357                       do_stat()      VFS
             fs/namev.c:45                        lookup()      VFS
             fs/namev.c:72                     dir_namev()      VFS
             fs/namev.c:90                   open_namev()       VFS
              fs/open.c:94                       do_open()      VFS
```

- make sure `PROCS=1`, `DRIVERS=1`, `VFS=1` in `Config.mk`



Kernel 2

Create devices, set current working directory

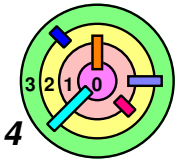
main/kmain.c:218	idleproc_run()	VFS
main/kmain.c:223	idleproc_run()	VFS

Reading from device and writing to device

fs/vnode.c:460	special_file_read()	VFS
fs/vnode.c:473	special_file_write()	VFS

Pathname resolution functions

fs/namev.c:45	lookup()	VFS
fs/namev.c:72	dir_namev()	VFS
fs/namev.c:90	open_namev()	VFS



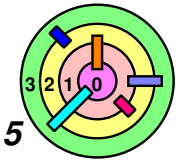
Kernel 2

System calls

fs/vfs_syscall.c:68	do_read()	VFS
fs/vfs_syscall.c:83	do_write()	VFS
fs/vfs_syscall.c:97	do_close()	VFS
fs/vfs_syscall.c:120	do_dup()	VFS
fs/vfs_syscall.c:136	do_dup2()	VFS
fs/vfs_syscall.c:168	do_mknod()	VFS
fs/vfs_syscall.c:189	do_mkdir()	VFS
fs/vfs_syscall.c:214	do_rmdir()	VFS
fs/vfs_syscall.c:235	do_unlink()	VFS
fs/vfs_syscall.c:263	do_link()	VFS
fs/vfs_syscall.c:278	do_rename()	VFS
fs/vfs_syscall.c:298	do_chdir()	VFS
fs/vfs_syscall.c:320	do_getdent()	VFS
fs/vfs_syscall.c:337	do_lseek()	VFS
fs/vfs_syscall.c:357	do_stat()	VFS

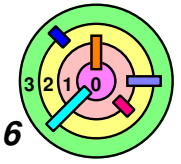
Open

fs/open.c:94	do_open()	VFS



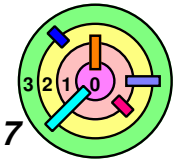
Kernel 2: Where To Start

- *(Phase 1)* Create directories and devices in `idleproc_run()`
  - `do_mkdir()` and `do_mknod()`
    - and whatever you need to support these functions (including *pathname resolution* functions in "`namev.c`")
      - `open_namev()`, `dir_namev()`, `lookup()`
  - maybe get `do_stat()` to work to check your work
- *(Phase 2)* Get kshell to work again in `initproc_run()`
  - kernel 2 kshell is different from kernel 1 kshell
    - kernel 2 kshell uses VFS!
  - need to get a few more functions to work so that you can interact with kshell
    - `do_open()`, `do_write()`, `do_read()`, `special_file_write()`, `special_file_read()`
- *(Phase 3)* Create *kshell command* to run "`vfstest`"
  - pass all tests in "`vfstest.c`"
  - also need to pass tests in "`faber_fs_test.c`"



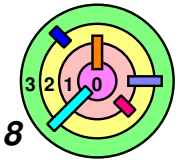
vnode.c

```
/*
 * If the file is a byte device then find the file's
 * bytedev_t, and call read on it. Return what read
 * returns.
 *
 * If the file is a block device then return -ENOTSUP
 */
static int
special_file_read(vnode_t *file, off_t offset,
                  void *buf, size_t count)
{
    NOT_YET_IMPLEMENTED("VFS: special_file_read");
    return 0;
}
```



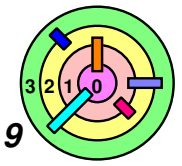
vnode.c

```
/*
 * If the file is a byte device find the file's
 * bytedev_t, and call its write. Return what write
 * returns.
 *
 * If the file is a block device then return -ENOTSUP.
 */
static int
special_file_write(vnode_t *file, off_t offset,
                   const void *buf, size_t count)
{
    NOT_YET_IMPLEMENTED("VFS: special_file_write");
    return 0;
}
```



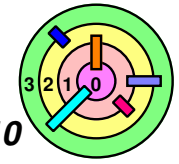
namev.c

```
/*
 * This takes a base 'dir', a 'name', its 'len', and a
 * result vnode.
 *
 * Most of the work should be done by the vnode's
 * implementation specific lookup() function.
 *
 * If dir has no lookup(), return -ENOTDIR.
 *
 * Note: returns with the vnode refcount on *result
 * incremented.
 */
int lookup(vnode_t *dir,
           const char *name,
           size_t len,
           vnode_t **result);
```



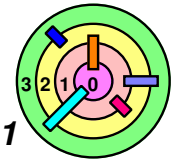
namev.c

```
/*
 * When successful this function returns data in the following
 * "out"-arguments:
 *  o res_vnode: the vnode of the parent directory of "name"
 *  o name: the `basename' (the element of the pathname)
 *  o namelen: the length of the basename
 *
 * For example: dir_namev("/s5fs/bin/ls", &namelen, &name, NULL,
 * &res_vnode) would put 2 in namelen, "ls" in name, and a pointer to
 * the vnode corresponding to "/s5fs/bin" in res_vnode.
 *
 * The "base" argument defines where we start resolving the path from:
 * A base value of NULL means to use the process's current working
 * directory, curproc->p_cwd.  If pathname[0] == '/', ignore base and
 * start with vfs_root_vn.  dir_namev() should call lookup() to take
 * care of resolving each piece of the pathname.
 *
 * Note: A successful call to this causes vnode refcount on *res_vnode
 * to be incremented.
 */
int dir_namev(const char *pathname,
              size_t *namelen,
              const char **name,
              vnode_t *base,
              struct vnode **res_vnode);
```



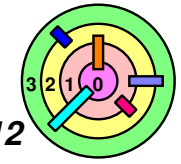
namev.c

```
/*
 * This returns in res_vnode the vnode requested by the
 * other parameters.  It makes use of dir_namev and
 * lookup to find the specified vnode (if it exists).
 * flag is right out of the parameters to open(2); see
 * <weenix/fcntl.h>.  If the O_CREAT flag is specified,
 * and the file does not exist call create() in the
 * parent directory vnode.
 *
 * Note: Increments vnode refcount on *res_vnode.
 */
int open_namev(const char *pathname,
               int flag,
               vnode_t **res_vnode,
               vnode_t *base);
```



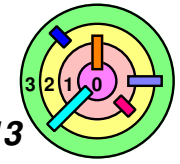
vfs_syscall.c

```
int do_close(int fd);
int do_read(int fd, void *buf, size_t nbytes);
int do_write(int fd, const void *buf, size_t nbytes);
int do_dup(int fd);
int do_dup2(int ofd, int nfd);
int do_mknod(const char *path, int mode, unsigned devid);
int do_mkdir(const char *path);
int do_rmdir(const char *path);
int do_unlink(const char *path);
int do_link(const char *from, const char *to);
int do_rename(const char *oldname, const char *newname);
int do_chdir(const char *path);
int do_getdent(int fd, struct dirent *dirp);
int do_lseek(int fd, int offset, int whence);
int do_stat(const char *path, struct stat *uf);
```



open.c

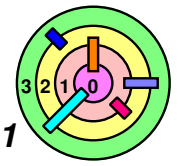
```
/*
 * There are a number of steps to opening a file:
 *      1. Get the next empty file descriptor.
 *      2. Call fget to get a fresh file_t.
 *      3. Save the file_t in curproc's file descriptor table.
 *      4. Set file_t->f_mode to OR of FMODE_(READ|WRITE|APPEND) based
 *         on oflags, which can be O_RDONLY, O_WRONLY or O_RDWR, possibly
 *         OR'd with O_APPEND or O_CREAT.
 *      5. Use open_namev() to get the vnode for the file_t.
 *      6. Fill in the fields of the file_t.
 *      7. Return new fd.
 *
 * If anything goes wrong at any point (specifically if the call to
 * open_namev fails), be sure to remove the fd from curproc, fput the
 * file_t and return an error.
 *
 * Error cases you must handle for this function at the VFS level:
 *      o EINVAL
 *      o EMFILE
 *      o ENOMEM
 *      o ENAMETOOLONG
 *      o ENOENT
 *      o EISDIR
 *      o ENXIO
 */
int do_open(const char *filename, int flags);
```



Kernel 3

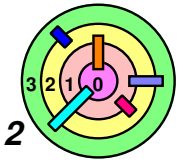
Bill Cheng

http://merlot.usc.edu/william/usc/



New Things in Kernel 3

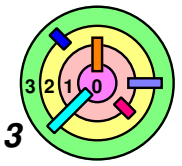
- Polymorphism go recursive!
  - fortunately, any recursion can be turned into a loop
    - so, think about the recursion as a loop
    - when you traverse a linked-list of shadow objects, you will eventually reach *bottom object* (and the recursion must stop)
- Address space
  - address space is implemented with *virtual memory maps*
    - review Ch 7 slides and *read kernel 3 FAQ*
- Handle page faults
  - keep Ch 7 slides in mind
- System call and system call support
  - `fork()` - where everything comes together
- Debugging
  - you may have to single-step machine instructions to find bugs



Kernel 3

```
% make nyi
```

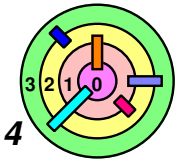
```
          mm/pframe.c:359          pframe_get()        VM
          mm/pframe.c:379          pframe_pin()        VM
          mm/pframe.c:395        pframe_unpin()        VM

         api/syscall.c:77            sys_read()        VM
         api/syscall.c:87           sys_write()        VM
        api/syscall.c:103         sys_getdents()        VM
         api/access.c:144            addr_perm()        VM
         api/access.c:160           range_perm()        VM
           proc/fork.c:77              do_fork()        VM
        vm/pagefault.c:72      handle_pagefault()        VM
           vm/shadow.c:73           shadow_init()        VM
           vm/shadow.c:85         shadow_create()        VM
           vm/shadow.c:97            shadow_ref()        VM
          vm/shadow.c:111            shadow_put()        VM
          vm/shadow.c:126     shadow_lookuppage()        VM
          vm/shadow.c:144       shadow_fillpage()        VM
          vm/shadow.c:153      shadow_dirtypage()        VM
          vm/shadow.c:160      shadow_cleanpage()        VM

       proc/kthread.c:161          kthread_clone()        VM
           fs/vnode.c:487      special_file_mmap()        VM
           fs/vnode.c:499 special_file_fillpage()        VM
           fs/vnode.c:511 special_file_dirtypage()        VM
           fs/vnode.c:523 special_file_cleanpage()        VM
```



Kernel 3

```
% make nyi
```

```
vm/vmmap.c:129          vmmap_create()          VM
vm/vmmap.c:138         vmmap_destroy()          VM
vm/vmmap.c:148          vmmap_insert()          VM
vm/vmmap.c:161       vmmap_find_range()         VM
vm/vmmap.c:171          vmmap_lookup()          VM
vm/vmmap.c:182          vmmap_clone()           VM
vm/vmmap.c:215            vmmap_map()           VM
vm/vmmap.c:251         vmmap_remove()           VM
vm/vmmap.c:262   vmmap_is_range_empty()         VM
vm/vmmap.c:277          vmmap_read()            VM
vm/vmmap.c:292         vmmap_write()            VM
   vm/brk.c:76               do_brk()           VM
  vm/anon.c:60           anon_init()             VM
  vm/anon.c:72         anon_create()             VM
  vm/anon.c:84            anon_ref()             VM
  vm/anon.c:98            anon_put()             VM
 vm/anon.c:106     anon_lookuppage()             VM
 vm/anon.c:115       anon_fillpage()             VM
 vm/anon.c:122      anon_dirtypage()             VM
 vm/anon.c:129      anon_cleanpage()             VM
  vm/mmap.c:55             do_mmap()             VM
  vm/mmap.c:70           do_munmap()             VM
```



/usr/bin/hello

Page frame management

mm/pframe.c:359	pframe_get()	VM
mm/pframe.c:379	pframe_pin()	VM
mm/pframe.c:395	pframe_unpin()	VM

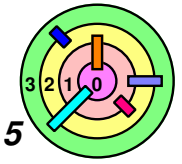
User address space implementation

vm/vmmap.c:129	vmmap_create()	VM
vm/vmmap.c:138	vmmap_destroy()	VM
vm/vmmap.c:148	vmmap_insert()	VM
vm/vmmap.c:161	vmmap_find_range()	VM
vm/vmmap.c:171	vmmap_lookup()	VM
vm/vmmap.c:182	vmmap_clone()	VM
vm/vmmap.c:215	vmmap_map()	VM
vm/vmmap.c:251	vmmap_remove()	VM
vm/vmmap.c:262	vmmap_is_range_empty()	VM
vm/vmmap.c:277	vmmap_read()	VM
vm/vmmap.c:292	vmmap_write()	VM

- some of them are needed by the loader in "`api/elf32.c`" since the loader creates an address space for a user program

Handle page faults

vm/pagefault.c:72	handle_pagefault()	VM



/usr/bin/hello

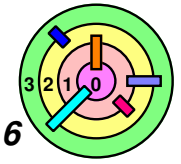
Basic system calls and system call support

api/syscall.c:77	sys_read()	VM
api/syscall.c:87	sys_write()	VM
api/syscall.c:103	sys_getdents()	VM
api/access.c:144	addr_perm()	VM
api/access.c:160	range_perm()	VM

Memory management (mmobj)

vm/anon.c:60	anon_init()	VM
vm/anon.c:72	anon_create()	VM
vm/anon.c:84	anon_ref()	VM
vm/anon.c:98	anon_put()	VM
vm/anon.c:106	anon_lookuppage()	VM
vm/anon.c:115	anon_fillpage()	VM
vm/anon.c:122	anon_dirtypage()	VM
vm/anon.c:129	anon_cleanpage()	VM

- recall that there are 3 types of `mmobj`
  - one lives inside the `vnode`
  - anonymous object
  - shadow object



/usr/bin/hello

- this may not be the complete list (I was told that these are what you need)

Basic system calls and system call support

api/syscall.c:77	sys_read()	
api/syscall.c:87	sys_write()	
api/syscall.c:103	sys_getdents()	
api/access.c:144	addr_perm()	VM
api/access.c:160	range_perm()	VM

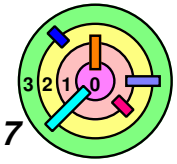
Memory management (mmobj)

"hello"

vm/anon.c:60	anon_init()	VM
vm/anon.c:72	anon_create()	VM
vm/anon.c:84	anon_ref()	VM
vm/anon.c:98	anon_put()	VM
vm/anon.c:106	anon_lookuppage()	VM
vm/anon.c:115	anon_fillpage()	VM
vm/anon.c:122	anon_dirtypage()	VM
vm/anon.c:129	anon_cleanpage()	VM

- recall that there are 3 types of `mmobj`
  - one lives inside the `vnode`
  - anonymous object
  - shadow object

To get "`hello`" to work, you need to get all of the above to work (except for `sys_read()` and `sys_getdents()`)



Beyond `/usr/bin/hello`

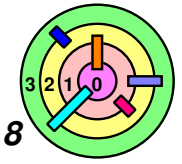
More memory management (mmobj)

vm/shadow.c:73	shadow_init()	VM
vm/shadow.c:85	shadow_create()	VM
vm/shadow.c:97	shadow_ref()	VM
vm/shadow.c:111	shadow_put()	VM
vm/shadow.c:126	shadow_lookuppage()	VM
vm/shadow.c:144	shadow_fillpage()	VM
vm/shadow.c:153	shadow_dirtypage()	VM
vm/shadow.c:160	shadow_cleanpage()	VM

- shadow object is complicated
  - only really need this when you get to run "`fork-and-wait`"

More system calls and system call support

proc/fork.c:77	do_fork()	VM
proc/kthread.c:161	kthread_clone()	VM
vm/mmap.c:55	do_mmap()	VM
vm/mmap.c:70	do_munmap()	VM
vm/brk.c:76	do_brk()	VM
fs/vnode.c:487	special_file_mmap()	VM
fs/vnode.c:499	special_file_fillpage()	VM
fs/vnode.c:511	special_file_dirtypage()	VM
fs/vnode.c:523	special_file_cleanpage()	VM



Beyond `/usr/bin/hello`

More memory management (mmobj)

vm/shadow.c:73	shadow_init()	VM
vm/shadow.c:85	shadow_create()	VM
vm/shadow.c:97	shadow_ref()	VM
vm/shadow.c:111	shadow_put()	VM
vm/shadow.c:126	shadow_lookuppage()	VM
vm/shadow.c:144	shadow_fillpage()	VM
vm/shadow.c:153	shadow_dirtypage()	VM
vm/shadow.c:160	shadow_cleanpage()	VM

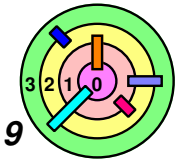
- shadow object is complicated
  - only really need this when you get to run "`fork-and-wait`"

More system calls and system call support

proc/fork.c:77	do_fork()	VM
proc/kthread.c:161	kthread_clone()	VM

---

vm/mmap.c:55	do_mmap()	VM
vm/mmap.c:70	do_munmap()	VM
vm/brk.c:76	do_brk()	VM
fs/vnode.c:487	special_file_mmap()	VM
fs/vnode.c:499	special_file_fillpage()	VM
fs/vnode.c:511	special_file_dirtypage()	VM
fs/vnode.c:523	special_file_cleanpage()	VM



Beyond `/usr/bin/hello`

More memory management (mmobj)

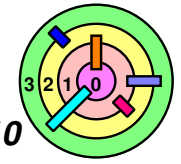
vm/shadow.c:73	shadow_init()	VM
vm/shadow.c:85	shadow_create()	VM
vm/shadow.c:97	shadow_ref()	VM
vm/shadow.c:111	shadow_put()	VM
vm/shadow.c:126	shadow_lookuppage()	VM
vm/shadow.c:144	shadow_fillpage()	VM
vm/shadow.c:153	shadow_dirtypage()	VM
vm/shadow.c:160	shadow_cleanpage()	VM

- shadow object is complicated
  - only really need this when you get to run "`fork-and-wait`"

More system calls and system call support

"`/sbin/init`"

proc/fork.c:77	do_fork()	VM
proc/kthread.c:161	kthread_clone()	VM
vm/mmap.c:55	do_mmap()	VM
vm/mmap.c:70	do_munmap()	VM
vm/brk.c:76	do_brk()	VM
fs/vnode.c:487	special_file_mmap()	VM
fs/vnode.c:499	special_file_fillpage()	VM
fs/vnode.c:511	special_file_dirtypage()	VM
fs/vnode.c:523	special_file_cleanpage()	VM

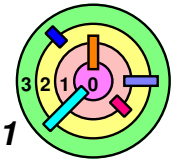


Strategy

- You must implement page frame management first

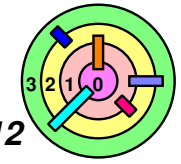
`mm/pframe.c:359`	`pframe_get()`	`VM`
`mm/pframe.c:379`	`pframe_pin()`	`VM`
`mm/pframe.c:395`	`pframe_unpin()`	`VM`

  - set `S5FS=1` but keep `VM=0` in `Config.mk` (even though this is kernel 3)
    - `VM=1` means that you are running user-space programs
  - the re-run `vfstest` and make sure everything works
    - `s5fs` has a real disk and `ramfs` didn't
  - don't bother with tests in "`faber_fs_test.c`" because they were designed for kernel 2 only
  - you should get this done by *end of week 12*
    - if code is mostly working but cannot past all `vfstest`, split up your team
      - one to debug code in "`pframe.c`", the rest move forward



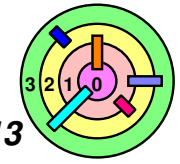
Strategy

- In the *first week* of kernel 3, you should get the *first user-space program* (i.e., "`hello`") to run to run
  - in `initproc_run()`, run "`/usr/bin/hello`" using `kernel_execve()`
    - the kernel's `INIT` process becomes the user-space "`hello`" process
    - you have to be able to *load* the program into memory
    - you have to be able to *build and manipulate* the *address space*
    - you have to be able to *handle page faults*
    - you have to get some *system call* (such as `write()`) to work
  - that's a lot of stuff to get working just to run "hello"
    - please do *not* even attempt to run "`/sbin/init`"
  - would be good to get this done by the *end of week 1 of kernel 3*



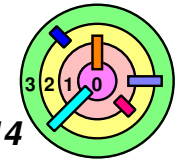
Strategy

- In the *2nd week* of kernel 3, you should get all the user-space programs in section (B) of the grading guidelines running from `initproc_run()`
  - the last one is "`/usr/bin/fork_and_wait`"
    - you need to get `fork()` to work
  - by the end of the weekend, you should get "`/sbin/init`" to run from `initproc_run()`
    - the kernel's `INIT` process becomes the user-space `INIT` process
      - the user-space `INIT` process spawn child processes to run user-space shell ("`/bin/sh`")
      - from the user-space shell ("`/bin/sh`"), you should be able to re-run all the section (B) user programs
  - would be good to get this done by the *end of weekend of the 2nd week of kernel 3*
- In the *last week* of kernel 3, get everything else to work



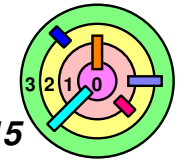
pframe.c

```
/*
 * Find and return the pframe representing the page identified by the
 * object and page number. If the page is already resident in memory,
 * then we return the existing page. Otherwise, we allocate a new page
 * and fill it (in which case this routine may block). Before allocating
 * the new pframe, we check to see if we need to call pageoutd and wake
 * it up if necessary.
 *
 * If the page is found (resident) but busy, then we will wait for it to
 * become unbusy and then try again (since it may have been freed after
 * that). Thus, as long as this routine returns successfully, the returned
 * page will be a non-busy page that will be guaranteed to remain resident
 * until the calling context blocks without first pinning the page.
 *
 * This routine may block at the mmobj operation level.
 *
 * @param o the parent object of the page
 * @param pagenum the page number of this page in the object
 * @param result used to return the pframe (NULL if there's an error)
 * @return 0 on success, < 0 on failure.
 */
int
pframe_get(struct mmobj *o, uint32_t pagenum, pframe_t **result)
{
    NOT_YET_IMPLEMENTED("VM: pframe_get");
    return 0;
}
```



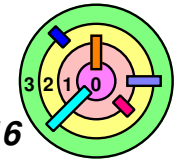
pframe.c

```
/*
 * Increases the pin count on this page. Pages with a pin count > 0 will
 * not be paged out by pageoutd, so this ensures that the page will remain
 * resident until the pin count is decreased.
 *
 * If the pframe has not yet been pinned, remove this pframe's list link
 * from the allocated list and add it to the pinned list.  Be sure to
 * decrement nallocated and increment npinned.
 *
 * In either case, increment the pf_pincount.
 *
 * @param pf the page to pin
 */
int
pframe_pin(pframe_t *pf)
{
    NOT_YET_IMPLEMENTED("VM: pframe_pin");
    return 0;
}
```



pframe.c

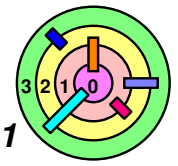
```
/*
 * Decreases the pin count on a page. If the pin count reaches zero,
 * then the page could be paged out any time after the calling context
 * blocks.
 *
 * If the pin count reaches zero, move the pframe's list link from the
 * pinned list to the allocated list.  Be sure to correctly update
 * npinned and nallocated
 *
 * @param pf a pinned page (a page with a positive pin count)
 */
int
pframe_unpin(pframe_t *pf)
{
    NOT_YET_IMPLEMENTED("VM: pframe_unpin");
    return 0;
}
```



More On Kernel 3

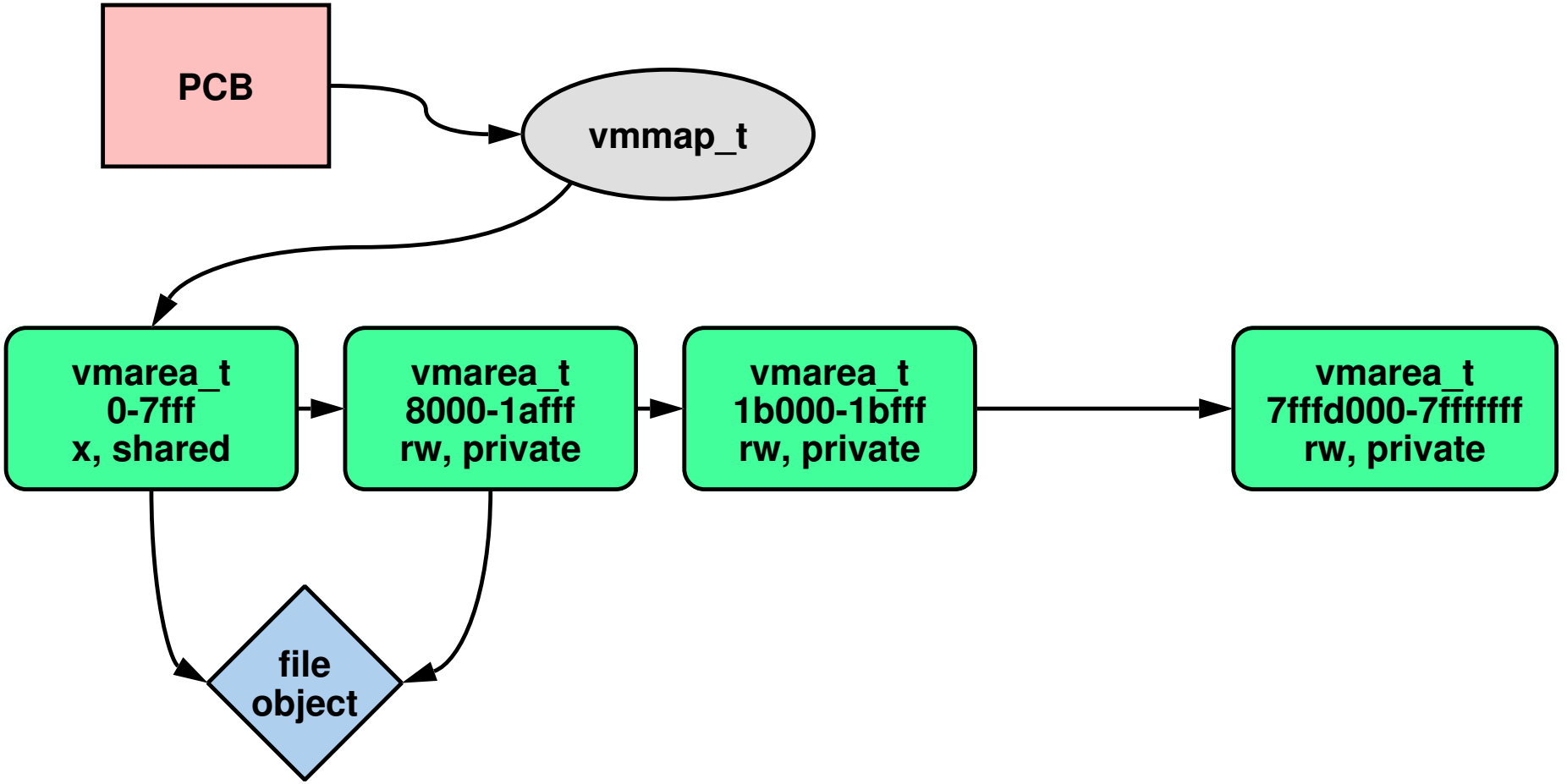
Bill Cheng

http://merlot.usc.edu/william/usc/

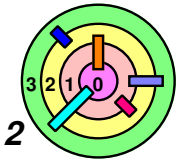


Address Space Implementation

- ***Address Space*** **is implemented using** ***Virtual Memory Map*** **(`vmmap`)**
  - **in lectures, sometimes I used the term "memory map"**

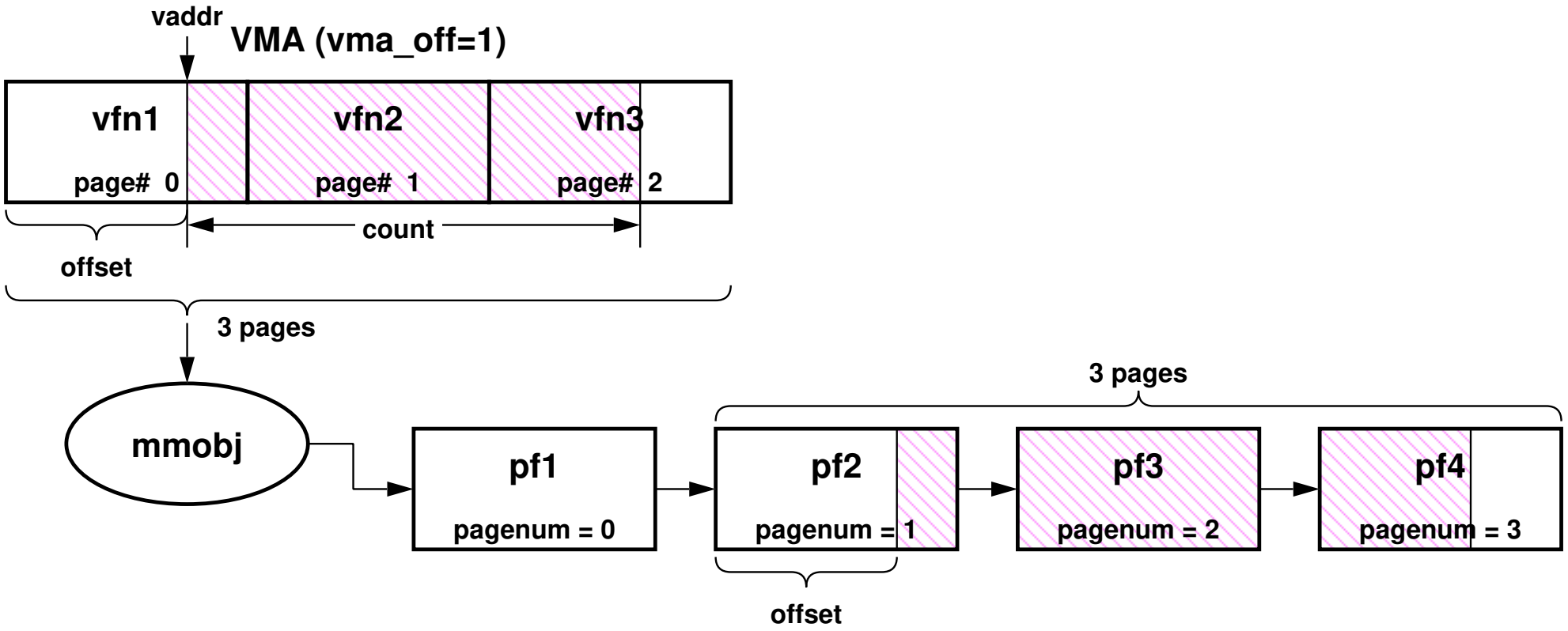


- **we will use `vma` or `vmarea` to refer to `vmarea_t`**
- **the values in `vmas` above are not what's in weenix**

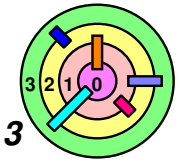


mmobj and pagenum

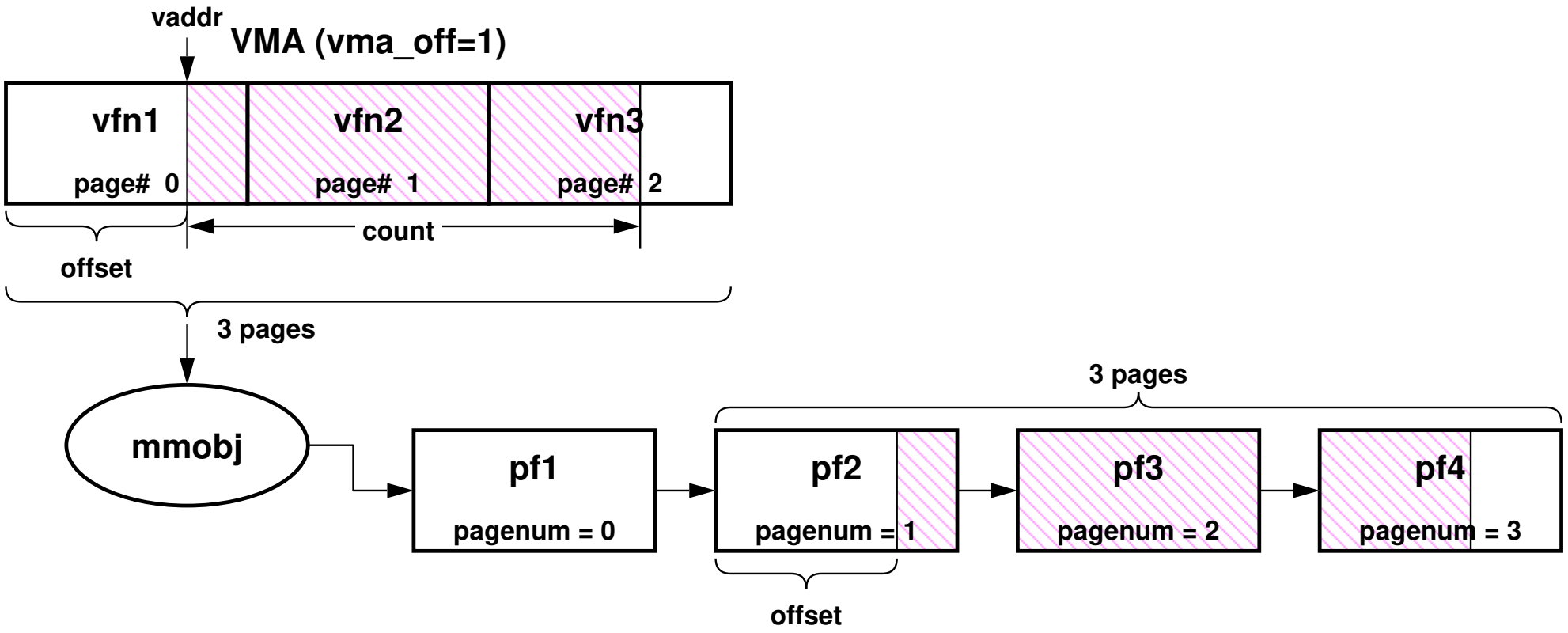
- This is a very important picture in the kernel FAQ to understand



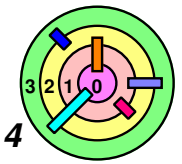
- read the kernel FAQ about "pagenum" and the next few FAQ items that follow it
- these are crucial in understanding how to build an address space for a user process



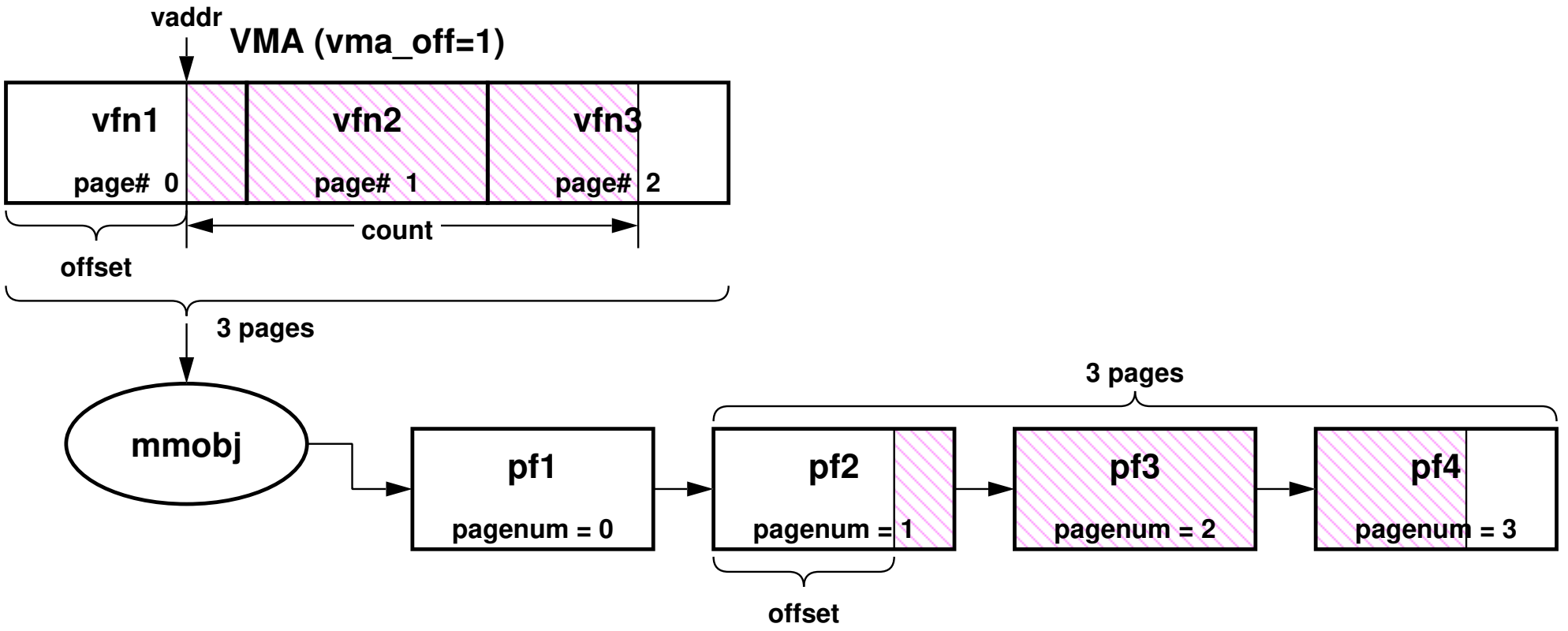
mmobj and pagenum



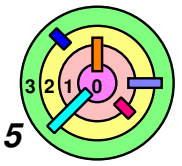
1) The address space is made up of a list of *non-overlapping* `vmareas`
2) Each `vmarea` is a memory segment (contiguous virtual memory locations)
   - in the above, it's shown as `[vaddr,vaddr+count)`
     - typically `vaddr` (first virtual address of a memory segment) is *page aligned*



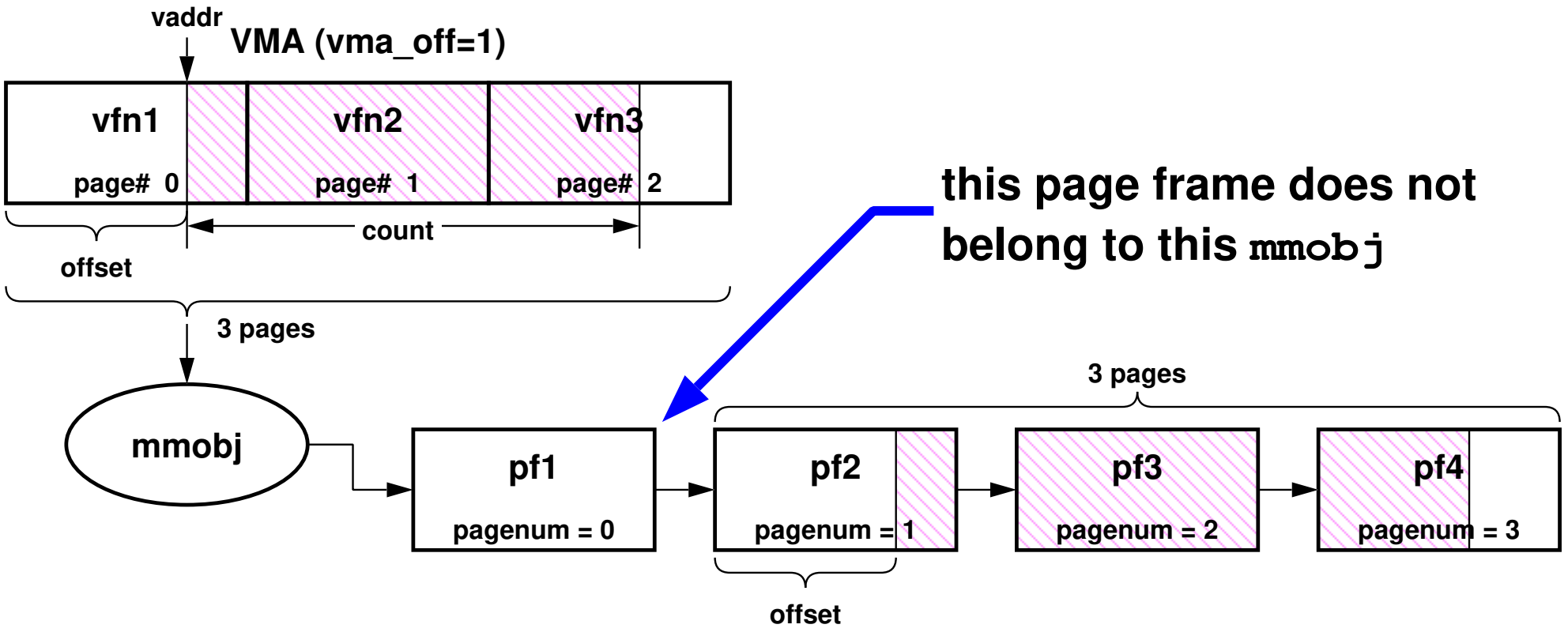
mmobj and pagenum



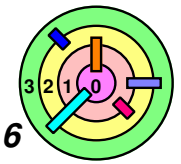
3) The kernel manages VM in "pages" (not "bytes"), it allocates enough pages so that a memory segment can fit inside
   - above, it takes 3 virtual pages to cover this memory segment
   - each virtual page need to be mapped into a physical page
     - `vfn` = `vpn` = virtual page number (20-bits long)
     - you need to get used to printing these numbers in ***hex***



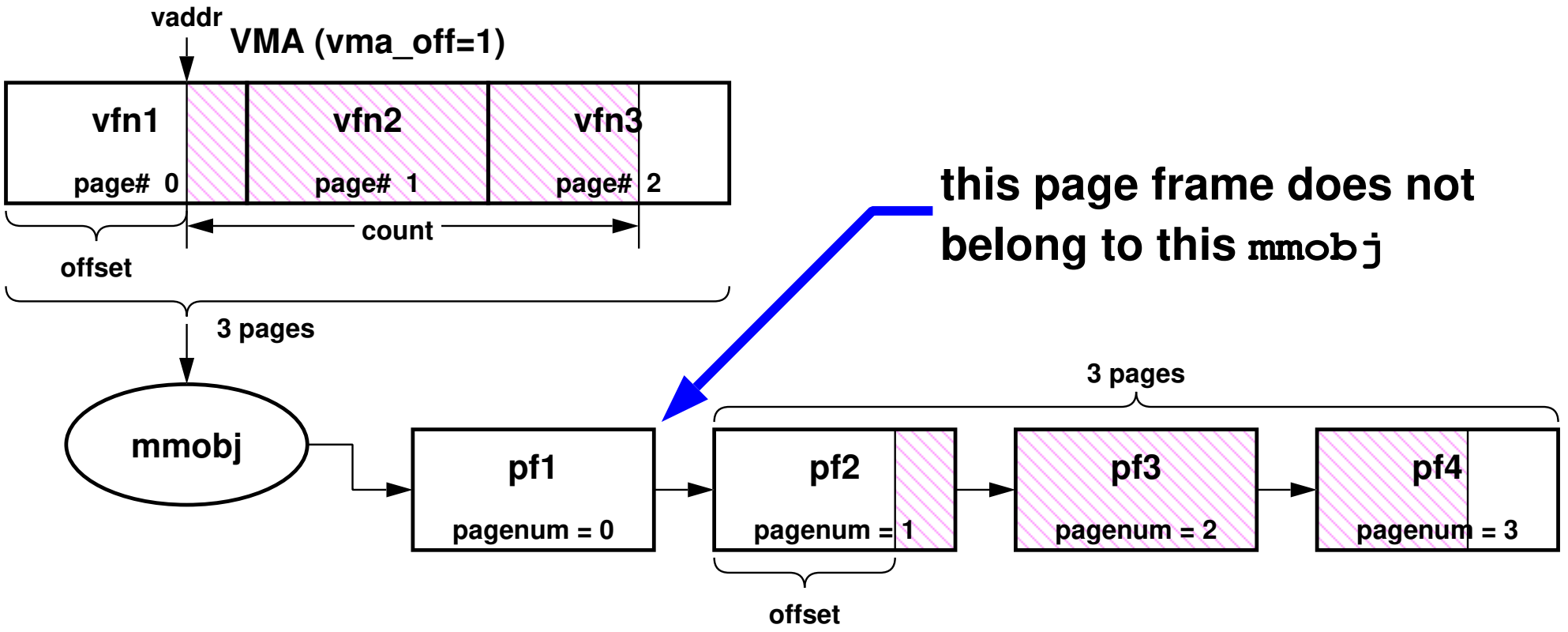
mmobj and pagenum



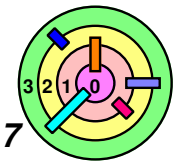
4) Page frames are managed by `mmobjs`
   - there is a physical page (hidden) inside each page frame
   - if an `mmobj` manages `N` page frames, their `pagenums` are `[vma_off, vma_off+N)` for that `mmobj`
   - *not really implemented as a linked list* as shown above
     - a *hash table* is used



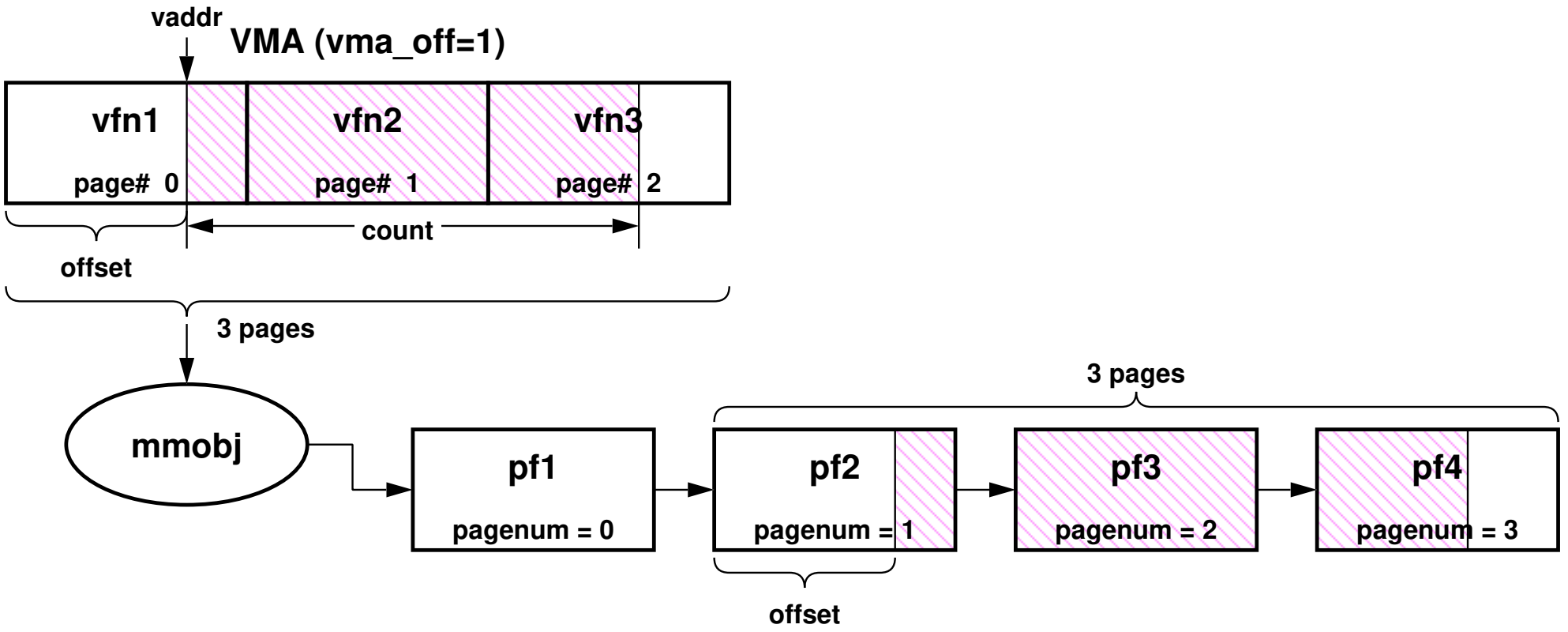
mmobj and pagenum



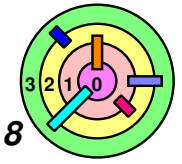
5) You can create multiple memory segments by mapping different pages of a file into your address space
   - conceptually, a file is divided into pages
   - `vma_off` in a memory segment gives you the starting page number of the file



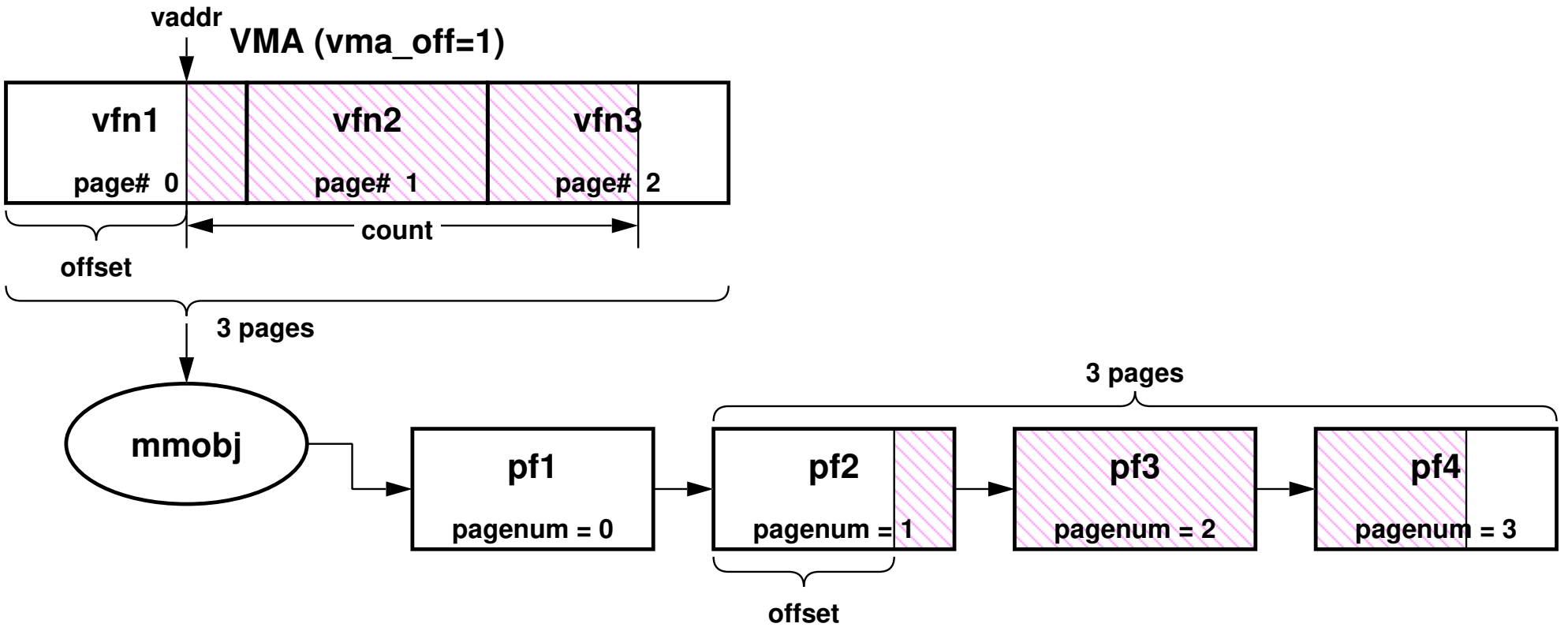
mmobj and pagenum



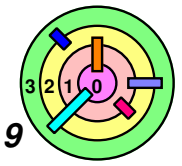
6) A page frame may not be present for an mmobj
   - since we are doing *demand paging*, in the beginning, none of the page frames are present for an mmobj
   - as page frames are brought in, they become *"resident"*
     - i.e., they are *cached* inside the corresponding mmobj
     - if modified, the page frame becomes *"dirty"*



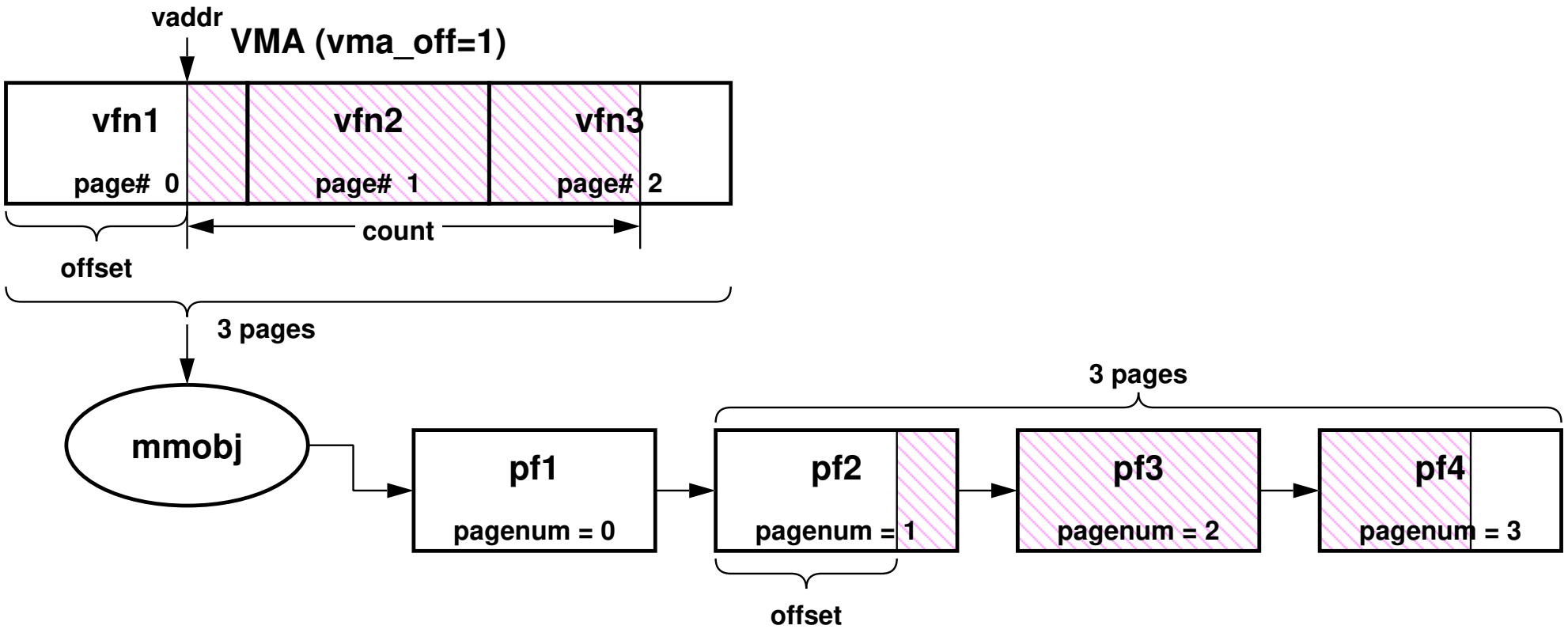
mmobj and pagenum



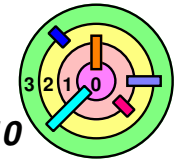
7) A page frame is identified by the `mmobj` that manages it and the `pagenum` of the page frame
   - in the kernel FAQ, it uses the notation *(o,n)* where *o* is an `mmobj` and *n* is a `pagenum`
   - a *linked list* of `mmobjs` must be used to make *copy-on-write* work correctly with `fork()`



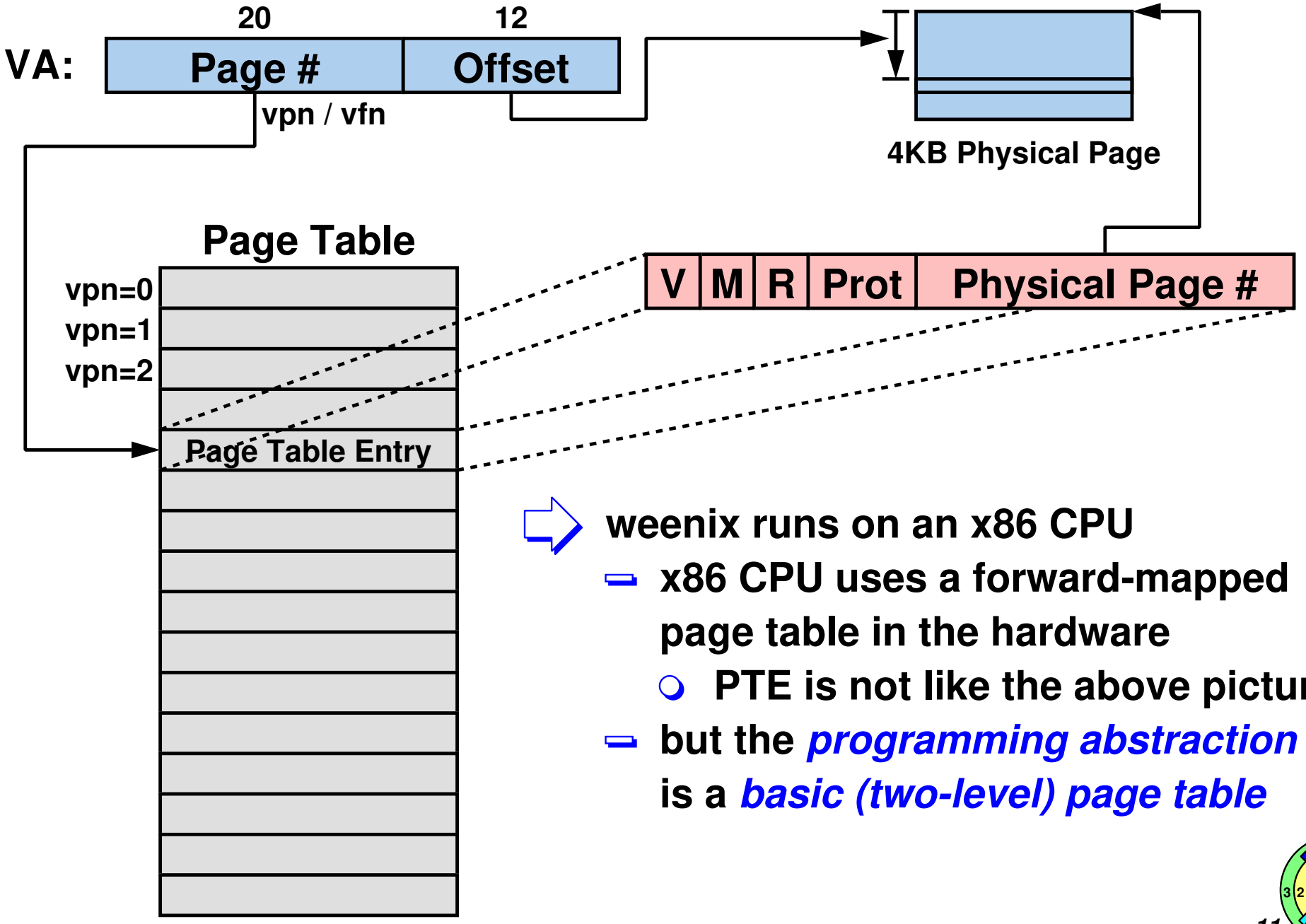
`mmobj` and `pagenum`



8) Page table maps a *virtual page number* to a *physical page number*
   - in the above picture, `vfn1`/`vfn2`/`vfn3` needs to be mapped to the physical page that lives inside `pf2`/`pf3`/`pf4`, respectively
   - `pf_addr` of a page frame contains a *kernel virtual address* for the corresponding "physical page"
     - `pt_virt_to_phys()` converts it to physical address



Forward-Mapped (Multilevel) Page Table



- weenix runs on an x86 CPU
  - x86 CPU uses a forward-mapped page table in the hardware
    - PTE is not like the above picture
  - but the *programming abstraction* is a *basic (two-level) page table*

Very Useful `gdb` Commands

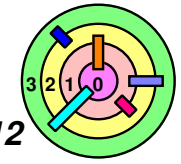
- Address space

```
kernel info vmmap_mapping_info curproc->p_vmmap
```

  - the *loader* builds address space for a user-space program
  - after your user-space program is "loaded", what does the address space looks like?
  - when you get your first legitimate page fault in `handle_pagefault()`, look at your address space
    - if it's is wrong, is there point proceeding?!
    - ask your classmates in the class Google Group if they are seeing the same thing
      - don't just ask others to share, that's not nice!

- Page table (not that useful)

```
kernel info pt_mapping_info curproc->p_pagedir
```



Read Some Code

- User
  - "hello.c"

```
int main(int argc, char **argv)
{
  open("/dev/tty0", O_RDONLY, 0);
  open("/dev/tty0", O_WRONLY, 0);
  write(1, "Hello, world!\n", 14);
  return 0;
}
```

- Kernel
  - "elf32.c" - the loader
  - "access.c" - need to understand `copy_from_user()` and `copy_to_user()`
  - "exec.c" - how to go into user space
  - "pagefault.c" - handle page fault
  - "vn_mmobj_ops.c" - code for the `mmobj` inside a `vnode`
  - "syscall.c" - to see how to implement `sys_write()`, look at how other `sys_*()` functions are implemented