```
1   CS 202, Spring 2020
2   Handout 11 (Class 16)
3
4   1. User-level threads and swtch()
5
6       We'll study this in the context of user-level threads.
7
8       Per-thread state in thread control block:
9
10          typedef struct tcb {
11              unsigned long saved_rsp;    /* Stack pointer of thread */
12              char *t_stack;              /* Bottom of thread's stack */
13              /* ... */
14              };
15
16      Machine-dependent thread initialization function:
17
18          void thread_init(tcb *t, void (*fn) (void *), void *arg);
19
20      Machine-dependent thread-switch function:
21
22          void swtch(tcb *current, tcb *next);
23
24      Implementation of swtch(current, next):
25
26          # gcc x86-64 calling convention:
27          # on entering swtch():
28          #  register %rdi holds first argument to the function ("current")
29          #  register %rsi holds second argument to the function ("next")
30
31          # Save call-preserved (aka "callee-saved") regs of 'current'
32          pushq %rbp
33          pushq %rbx
34          pushq %r12
35          pushq %r13
36          pushq %r14
37          pushq %r15
38
39          # store old stack pointer, for when we swtch() back to "current" later
40          movq %rsp, (%rdi)                    # %rdi->saved_rsp = %rsp
41          movq (%rsi), %rsp                    # %rsp = %rsi->saved_rsp
42
43          # Restore call-preserved (aka "callee-saved") regs of 'next'
44          popq %r15
45          popq %r14
46          popq %r13
47          popq %r12
48          popq %rbx
49          popq %rbp
50
51          # Resume execution, from where "next" was when it last entered swtch()
52          ret
53
54
```
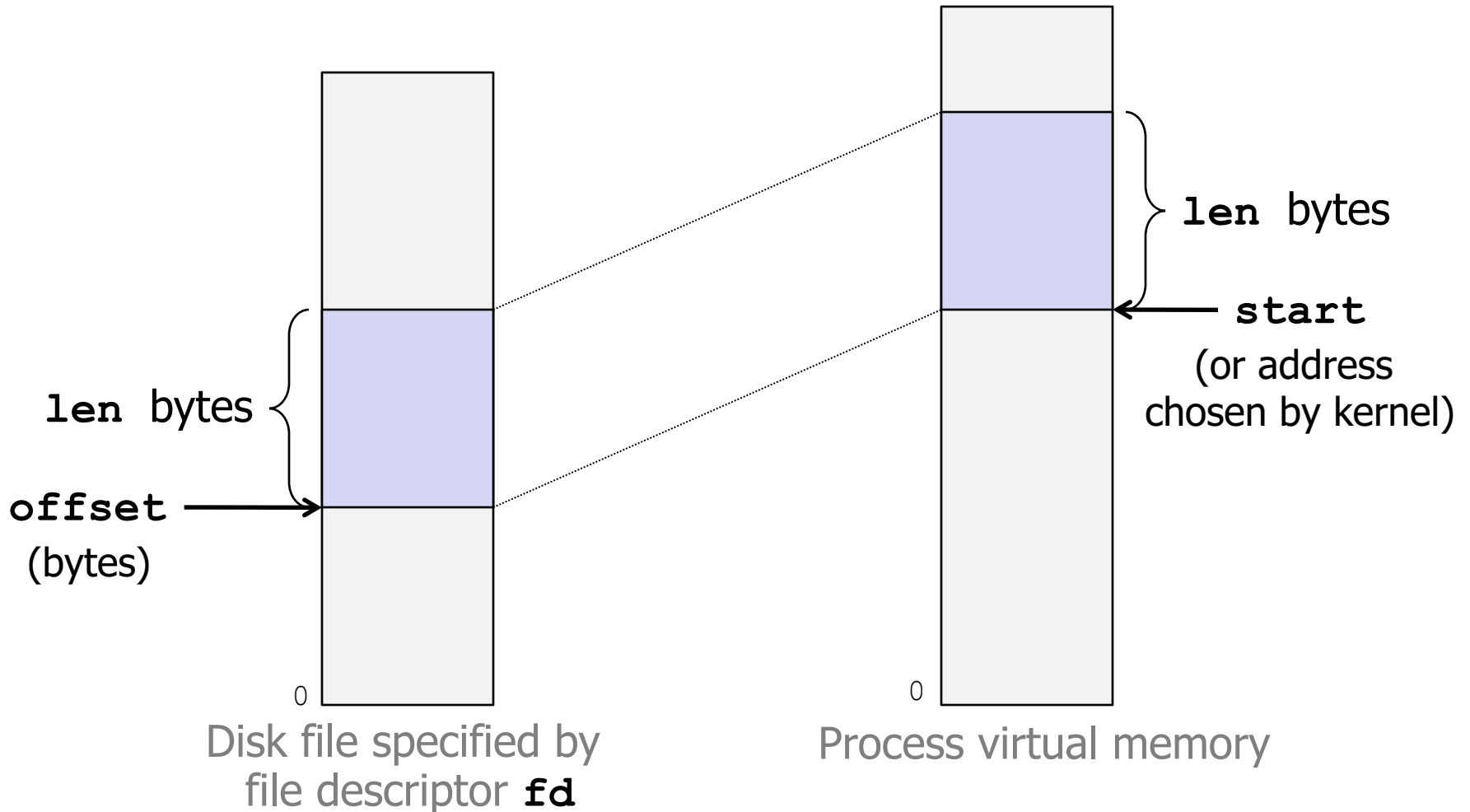
```
55
56  2. Example use of swtch(): the yield() call.
57
58      A thread is going about its business and decides that it's executed for
59      long enough. So it calls yield(). Conceptually, the overall system needs
60      to now choose another thread, and run it:
61
62      void yield() {
63
64          tid next    = pick_next_thread();  /* get a runnable thread */
65          tid current = get_current_thread();
66
67          swtch(current, next);
68
69          /* when 'current' is later rescheduled, it starts from here */
70      }
71
72  3. How do context switches interact with I/O calls?
73
74      This assumes a user-level threading package.
75
76      The thread calls something like "fake_blocking_read()". This looks
77      to the _thread_ as though the call blocks, but in reality, the call
78      is not blocking:
79
80      int fake_blocking_read(int fd, char* buf, int num) {
81
82          int nread = -1;
83
84          while (nread == -1) {
85
86              /* this is a non-blocking read() syscall */
87              nread = read(fd, buf, num);
88
89              if (nread == -1 && errno == EAGAIN) {
90                  /*
91                   * read would block. so let another thread run
92                   * and try again later (next time through the
93                   * loop).
94                   */
95                  yield();
96              }
97          }
98
99          return nread;
100     }
101
102
103
104
105
```

User-Level Memory Mapping

```
void *mmap(void *start, int len,
           int prot, int flags, int fd, int offset)
```

len bytes

start
(or address
chosen by kernel)

len bytes

offset
(bytes)

Disk file specified by
file descriptor **fd**

Process virtual memory

0

0

```c
1    #include <fcntl.h>
2    #include <stdio.h>
3    #include <stdlib.h>
4    #include <sys/mman.h>
5    #include <sys/stat.h>
6    #include <sys/types.h>
7    #include <unistd.h>
8
9    void mmapcopy(int fd, int size);
10
11   int main(int argc, char **argv) {
12     struct stat stat;
13     int fd;
14
15     /* Check for required cmd line arg */
16     if (argc != 2) {
17       printf("usage: %s <filename>\n", argv[0]);
18       exit(0);
19     }
20
21     /* Copy input file to stdout */
22     if ((fd = open(argv[1], O_RDONLY, 0)) < 0)
23       perror("open");
24
25     fstat(fd, &stat);
26     mmapcopy(fd, stat.st_size);
27
28     close(fd);
29
30     return 0;
31   }
32
33   void mmapcopy(int fd, int size) {
34
35     /* Ptr to memory mapped area */
36     char *bufp;
37
38     bufp = mmap(NULL, size, PROT_READ, MAP_PRIVATE, fd, 0);
39
40     write(STDOUT_FILENO, bufp, size);
41
42     return;
43   }
```