System Programming

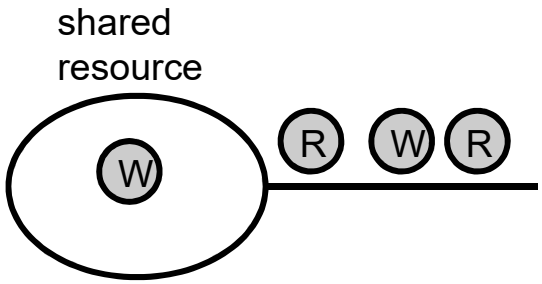
7. Record Lock

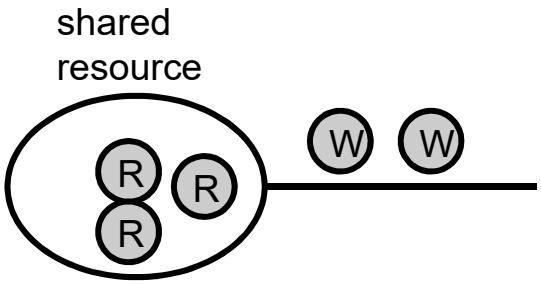
Seung-Ho Lim

Dept. of Computer & Electronic Systems Eng.

Concurrent-Readers / Exclusive-Writers

- **Problem Definition**
  - Some ***writer & reader processes*** access a shared resource (variables, files) concurrently.
  - ***Mutual exclusion*** must be provided to readers & writers
    - one by one accesses → low performance
    - Readers can share a shared resource because they do not modify it !
  - An improvement
    - Any writer process must access the shared resource exclusively from other writers and readers. ***(writing -> exclusive lock)***
    - But when reading, a reader must be protected from other writer processes, however, other readers can join in concurrent reading. (to enhance the performance! 80% of DB operations are readings!) ***(reading -> shared lock)***





Concurrent-Readers/ Exclusive-Writers Problem

- **Writer's lock (Exclusive lock)**
  - ***Mutex*** for writers
  - Same as the original ***mutex***
  - When writing, the writing is protected from all other writers & readers
- **Reader's lock (Shared lock)**
  - A special ***mutex*** for readers.
  - At the time when a reader requests the reader's lock, the reader can join if previous readers are currently accessing the shared resource. (reader's lock is already set) -> shares the reader's lock
  - At the time when a reader requests the reader's lock, the reader must wait if a previous writer is accessing the shared resource. (writer's lock is already set)

Record Lock in a File (1)

```
#include <fcntl.h>

int fcntl(int fildes, int cmd, struct flock *lock);
```

- parameters
  - fildes : file descriptor
  - cmd : command
    - F_GETLK : check if the lock can be acquired
      - » if already locked by someone, return a filled lock structure
      - » if it can be acquired, return a lock structure with F_UNLCK (l_type)
    - F_SETLK : try to set lock as designated in lock argument.
      - » called with l_type == F_RDLCK or F_WRLCK
      - » if already locked, return -1 immediately
      - » on an error, errno = EACCESS or EAGAIN

Record Lock in a File (2)

- **F_SETLKW** :
  - » blocking version for F_SETLK
  - » if already locked, the calling process must be blocked until it can get the lock.
- F_SETLK and l_type == F_UNLCK
  - » release a lock designated by the flock structure

Record Lock in a File (3)

```
struct flock {
   .......
   short        l_type; // type of lock: F_RDLCK, F_WRLCK, F_UNLCK
                // F_RDLCK : reader's lock
                // F_WRLCK : writer's lock
                // F_UNLCK : unlock
   short        l_whence;   // SEEK_SET, SEEK_CUR, SEE_END

   off_t        l_start;    // lock start position
   off_t        l_len;      // lock region length (0 for entire file)
   pid_t        l_pid:       // pid of a process that has a lock
                             // (used only in F_GETLK)
}
```

Example: Record Processing (1)

rec-proc.c

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>

# define NUM_RECORDS 100

struct record {
   char name[20];       // account owner
   int  id;             // account number
   int  balance;
};

void get_new_record(struct record *curr);
void display_record(struct record *curr);
```

Example: Record Processing (2)

```
int main( int argc, char *argv[])
{  FILE *fp; struct record current;
  int record_no;
   int fd;
   long pos;
   char yes;

  fd = open( argv[1], O_RDWR | O_CREAT , S_IRUSR | S_IWUSR);

   if(( fp = fdopen( fd, "r+")) == NULL)
   {    perror( argv[1]);
        exit(2);
   }
   printf("enter record number: ");
   scanf("%d",&record_no);
   getchar();
```

Example: Record Processing (3)

```
   while (record_no >= 0 && record_no < NUM_RECORDS) {
        pos = record_no * sizeof(struct record);
        fseek( fp, pos, SEEK_SET);
        fread(&current, sizeof(struct record), 1, fp);
        display_record (&current);
        printf("update records? yes = y\n");
        scanf("%c", &yes); getchar();
        if (yes == 'y') {
            get_new_record(&current);
            fseek( fp, pos, SEEK_SET);
            fwrite( &current, sizeof( struct record), 1, fp);
            printf("update done\n");
        }
        printf("enter record number: ");
        scanf("%d",&record_no);
        getchar();
   }
   fclose(fp);
}
```

Example: Record Processing (4)

```
 void get_new_record(struct record *curr)
{
    printf("> id? ");
    scanf("%d", &curr->id);
    printf("> name? ");
    scanf("%s", curr->name);
    printf("> balance? ");
    scanf("%d", &curr->balance);
}

void display_record(struct record *curr)
{
    printf("\n");
    printf("id: %d \n", curr->id);
    printf("name: %s \n", curr->name);
    printf("balance: %d \n", curr->balance);
    printf("\n");
}
```

Is this code working good?

Critical Section

Process A

```
read balance;
update balance(deposit 5);
write balance;
```

Process B

```
read balance;
update balance(deposit 6);
write balance;
```

Think the following scenario!

read by Process A; (current balance: 10)

read by Process B; (current balance: 10)

write by Process A; (update balance: 15)

write by Process B; (update balance: 16)

Record Locking

- Deposit / withdraw (*update balance*)
  - This (write) operation must be done exclusively with other deposit and inquiry operations. This is called a ***writer's lock*** or an ***exclusive lock.***
- Inquiry (*read balance*)
  - This operation can be done with other inquiry operations but must be done exclusively with other write operations.
  - This is called a ***reader's lock*** or a ***shared lock.***

		Request for read lock	Request for write lock
Region currently has	No locks	OK	OK
	One or more read locks	OK	denied
	One write lock	denied	denied

Pseudo Code

```
open the master account file;
while (true) {
   get the operation type and information on the account
   switch (operation) {
            case create:
                     get a writer's lock on the record;
                     get id and name of the user;
                     reset the account information;
                     release the writer's lock;
            case inquiry:
                      get a reader's' lock on that record;
                     (if a write lock exists, blocked)
                     display the account information;
                     release the reader's lock;
            case deposit (or withdraw):
                     get a writer's lock on that record;
                     (if a writer's or a reader's lock exists, blocked)
                     display the account information;
                     release the writer's lock;
   }
```

Example (1)

reclock.c

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>

#define TRUE    1
#define FALSE   0

#define NUM_RECORDS 100

struct record{
   char name[20];     // account owner
   int id;            // account umber
   int balance;       // balance
};

int reclock (int fd, int recno, int len, int type);
void display_record(struct record *curr);
```

Example (2)

```
int main()
{
    struct record current;
    int record_no;
    int fd, pos, i, n;
    char yes;
    char operation;
    int amount;
    char buffer[100];
    int quit=FALSE;

    fd = open("./account", O_RDWR | O_CREAT, S_IRUSR | S_IWUSR);
    while (1) {
          printf("enter account number (0-99): ");
          scanf("%d", &record_no);
          fgets(buffer, 100, stdin);

          if (record_no < 0 && record_no >= NUM_RECORDS)
                    break;
          printf("enter operation name (c/r/d/q): ");
          scanf("%c", &operation);
```

Example (3)

```
switch (operation) {
      case 'c' : // create
                 reclock(fd, record_no, sizeof(struct record),F_WRLCK);
                 pos = record_no * sizeof(struct record);
                 lseek(fd, pos, SEEK_SET);
                 printf("> id ? ");
                 scanf("%d", &current.id);
                 printf("> name ? ");
                 scanf("%s", current.name);
                 current.balance = 0 ;
                 n = write(fd, &current, sizeof(struct record));
                 display_record (&current);
                 reclock(fd, record_no, sizeof(struct record), F_UNLCK);
                 break;
      case 'r' :          // inquiry
                 reclock(fd, record_no, sizeof(struct record), F_RDLCK);
                 pos = record_no * sizeof(struct record);
                 lseek(fd, pos, SEEK_SET);
                 n  = read(fd, &current, sizeof(struct record));
                 display_record (&current);
                 reclock(fd, record_no, sizeof(struct record), F_UNLCK);
                 break;
```

Example (4)

```
            case 'd' :          // deposit
                        reclock(fd, record_no, sizeof(struct record), F_WRLCK);
                        pos = record_no * sizeof(struct record);
                        lseek(fd, pos, SEEK_SET);
                        n = read(fd, &current, sizeof(struct record));
                        display_record (&current);
                        printf("enter amount\n");
                        scanf("%d", &amount);
                        current.balance += amount;
                        lseek(fd, pos, SEEK_SET);
                        write(fd, &current, sizeof(struct record));
                        reclock(fd, record_no, sizeof(struct record), F_UNLCK);
                        break;
            case 'q' :
                        quit = TRUE;
                        break;
            default :
                        printf("illegal input\n");
                        continue;
            }
    }
    close(fd);
    fflush(NULL);
}
```

Example (5)

```
int reclock (int fd, int recno, int len, int type)
{
     struct flock fl;
     switch (type) {
            case F_RDLCK:
            case F_WRLCK:
            case F_UNLCK:
                        fl.l_type = type;
                        fl.l_whence = SEEK_SET;
                        fl.l_start = recno * len;
                        fl.l_len = len;
                        fcntl (fd, F_SETLKW, &fl);
                        return 1;
            default:
                        return -1;
    }
}
void display_record(struct record *curr)
{
     printf("\n");
     printf("id: %d \n", curr->id);
     printf("name: %s \n", curr->name);
     printf("balance: %d \n", curr->balance);
     printf("\n");
}
```

File Lock

```
#include <fcntl.h>

int flock(int fd, int operation);
```

- locks entire file (not a record or a part)
- parameters
  - *fd* : open file descriptor
  - operation
    - LOCK_SH: place a shared lock
    - LOCK_EX: place an exclusive lock
    - LOCK_UN: remove an existing lock held by this process
    - can be OR'd with LOCK_NB (non-blocking)

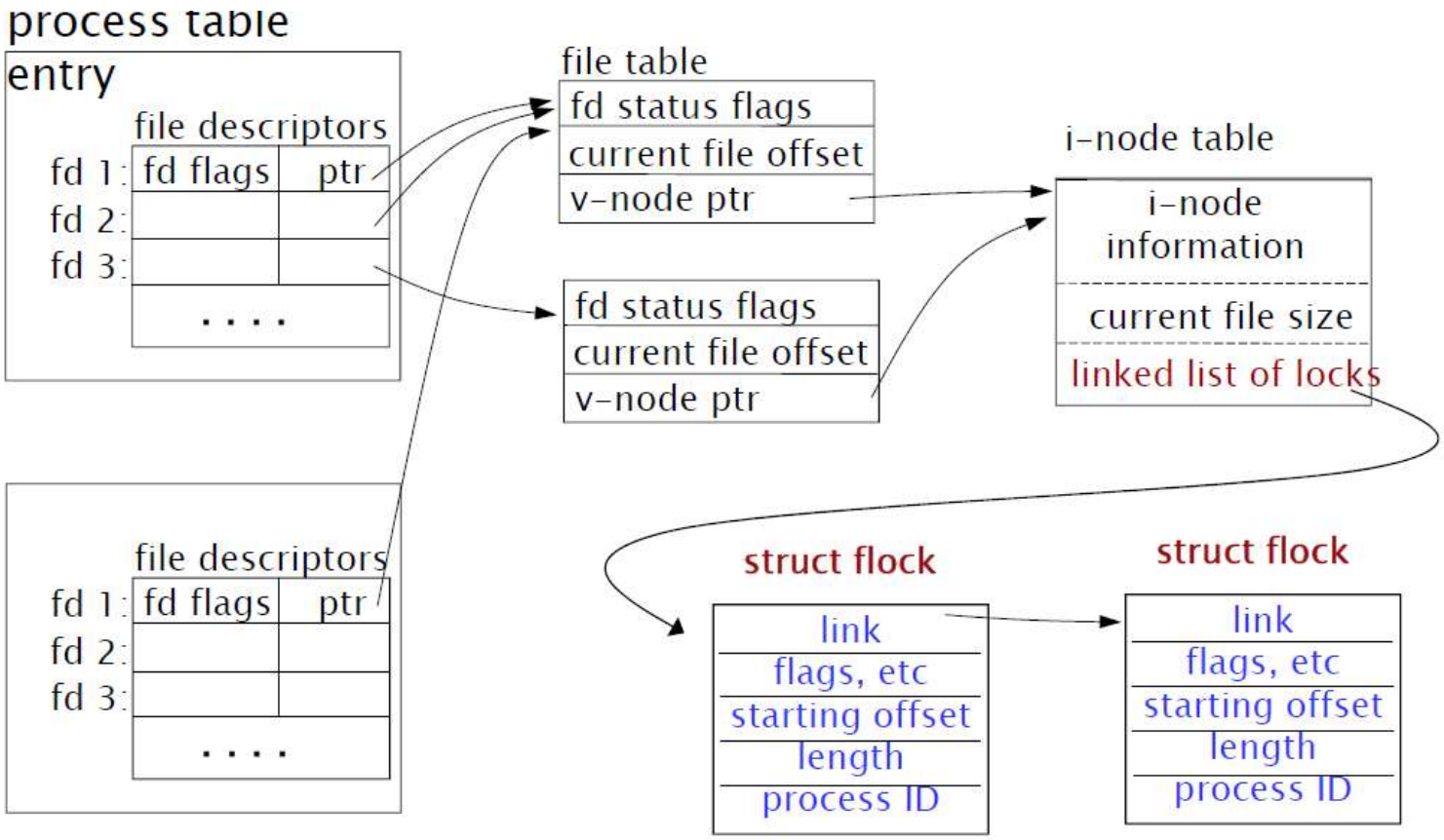
Lock inheritance & release

- Inheritance
  - a lock is not inherited to a child process by fork()
  - a lock is inherited to a process by exec()
- Release
  - When a process terminates, all locks created by the process will be released
  - When a file descriptor to a file is close, all locks of the file is released

```
fd1 = open("a", … );
read_lock(fd1, … );
fd2 = dup(fd1);
close(fd2)
```

```
fd1 = open("a", … );
read_lock(fd1, … );
fd2 = open("a",… );
close(fd2);
```

Flocks in a kernel



- lock is associated with a file (not a fd)
- if any one of fd1, fd2, and fd3 is closed, all the locks will be released

Advisory vs. Mandatory Locking

- Advisory locking
  - locking is not enforced by a kernel
  - reads or writes can violate the locking protocol
  - processes should voluntarily conform the locking protocol
- Mandatory locking
  - kernel enforces the locking protocol
  - any read or write can not violate the locking protocol
  - kernel overhead is quite high since it should inspect every read and write call
  - how to set the mandatory lock bit
    - turn ON the set-group-ID bit and OFF group-execute bit

    e.g.

    $ chmod 2644 lockfile (or chmod g+s,g-x lockfile)

    ```
    -rw-r-lr-- 1 rheeys 5 Jul 18 12:11 lockfile
    ```

Ex1

- Producer & Consumer in Pthreads :
  - In the lecture note, an array has been used as the buffer between the producer & consumer threads.
  - We provide linked list, the "*struct LinkedList*" has
    - *Node Head* & *tail* pointers, *number_of_items,*
    - *Node : integer data, Node pointer to next*
    - intertItem: at head, if *number_of_items* >= 100, then wait,
    - getItem: at tail, if *number_of_items* == 0, then wait,
    - isEmpty: check number_of_items == 0
    - isFull: check number_of_items == 100

Ex1

- Producer & Consumer in Pthreads :
  - Modify the program as follows:
    - Buffer array -> integer buffer queue in a linked list.
    - Share linked list between producer & consumer
    - Synchronization with mutex & condition variable
    - Producer
      - » If the linked list is full then wait,
      - » Insert random integer data at the tail of the list if possible.
      - » usleep some amount of time
    - Consumer
      - » If the linked list is empty then wait,
      - » Remove and get the data at the head of the list if possible.
      - » print the data to the console.
      - » usleep some amount of time

EX2

- File RW lock handling
  - Add “withdraw” & “transfer” functions to the example ATM program.
  - Commands
    - Existing command : (c/r/d/q)
    - Add command w,t
    - “Withdraw”, “Deposit”
      - » Use a writer’s lock on the record.
    - “Transfer”
      - » Withdraw + deposit must be done atomically.
      - » So
        - » Writer’s lock on the withdraw account;
        - » Writer’s lock on the deposit account;
        - » Do withdraw & deposit;
        - » Unlock all.

Due Date

- Until 5/7
- Submit to eclass