19CS2106S, 19CS2106A Test - I

Set – 1

Key and Scheme

1. **Disk blocks allocated to a file are added to the free list when the file is deleted. Write an algorithm to perform this operation in Unix.**

   Scheme of Evaluation:

   Explanation/diagram - 2 M

   Algorithm - 2.5 M modification of alloc Algorithm for Allocating Disk Block.

   Maurice J. Bach, The Design of The Unix Operating System, 2013 PHI Publishing, section:4.8, Page number: 87

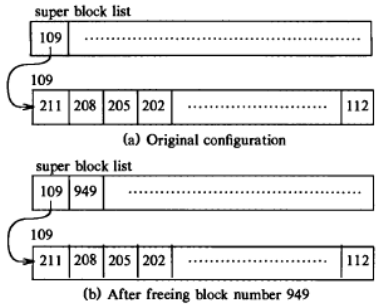


Figure 4.20. Freeing Disk Blocks

2. **xv6's mkfs program generate layout for an empty file system. Illustrate xv6 on-disk layout where Block 0, 1, 2 are fixed. Explain the purpose and goal of Logging (Transactions).**

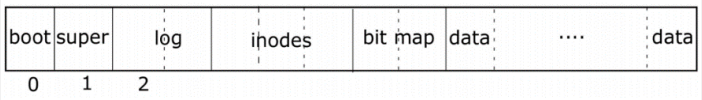
Scheme of Evaluation:

Illustrate xv6 on-disk layout where Block 0, 1, 2 are fixed. 2.5 M

Explain the purpose and goal of Logging (Transactions). 2M

Russ Cox, Frans Kaashoek, Robert Morris, xv6: a simple, Unix-like teaching operating system", Revision https://pdos.csail.mit.edu/6.828/2018/xv6/book-rev11.pdf, Chapter: 6, Page number: 77

xv6 layout of File System



Structure of the xv6 file system. The header fs.h contains constants and data structures describing the exact layout of the file system.

Notice: Block 0, 1, 2 are fixed

Block 0: Boot code

Block 1: Super Block. Store metadata about the file system

- Size (# of the blocks)
- # of data blocks
- # of inodes
- # of blocks in log
- Besides, super block also fills in by a small program called **mkfs (mkfs.c)** which is an initial file system

Block 2: Log area. Use for **transactions**. Maintain consistency in case of a power outrage or system shutdown accidentally.

Logging (Transactions)

Purpose: Do transactions to achieve crash recovery. Transactions mean group multiple writes into one bundle

Goal: For **Consistency**

3. **(i). What is the buffer header and during system initialization why kernel allocates space for a number of buffers. Suppose the kernel does a delayed write of a block. What happens when another process takes that block from its hash queue? From the free list?**

   (ii). Write a system program that list files names and inode numbers in a given directory, like when you execute the following command:

   $ ls – ia

   Scheme of Evaluation:

   Buffer Header - 2

   Delayed write of a block and race condition - 2 M

   A system program - Uses readdir to populate a dirent structure - 4 M

   Maurice J. Bach, The Design of The Unix Operating System, 2013 PHI Publishing, Section 3.3

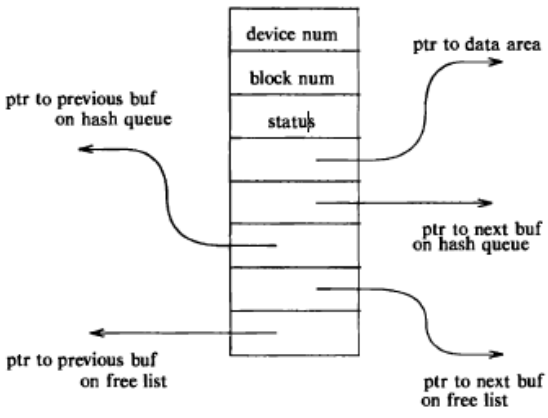
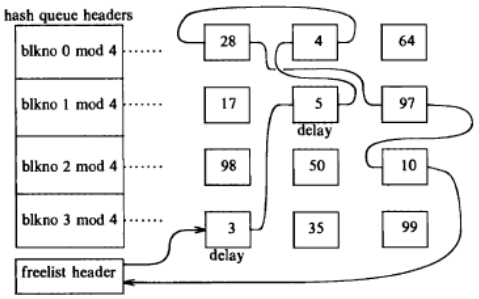


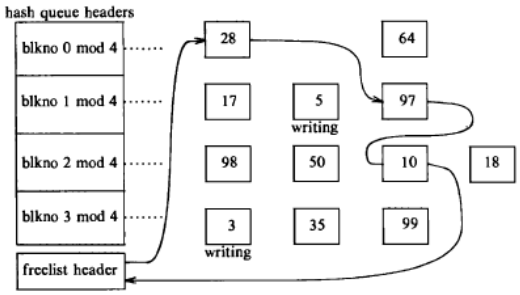
Figure 3.1. Buffer Header

/* Program: lls.c -- Uses readdir to populate a dirent structure */

```
#include <dirent.h> /* For DIR and struct dirent */
#include <stdio.h>
#include "quit.h"
int main(int argc, char **argv) {
DIR *dir; /* Returned by opendir */
struct dirent *direntry; /* Returned by readdir */
arg_check(2, argc, "Specify a directory\n", 1) ;
if ( (dir = opendir(argv[1])) == NULL) /* Directory
must exist and */
{ perror("opendir"); /* have read permission */
   exit(1);}
while ((direntry = readdir(dir)) != NULL) /* Until
entries are exhausted */
printf("%10d %s\n", direntry->d_ino, direntry-
>d_name);
closedir(dir);
exit(0);
}
```



(a) Search for Block 18, Delayed Write Blocks on Free List



(b) Writing Blocks 3, 5, Reassign 4 to 18

Figure 3.8. Third Scenario for Buffer Allocation

4. (i). When can a file be deleted from disk? how does xv6 delete a file? List xv6 kernel code functions/algorithms and files directly or indirectly used for command $ rm a
(ii). How read system calls work. Explain algorithm. What are its input parameters and returns information? Describe xv6 functions: filealloc, filedup, and fileclose.

When can a file be deleted from disk? – 2M

An inode has two counts associated with it: nlink says how many links point to this file in the directory tree. The count ref that is stored only in the memory version of the inode counts how many C pointers exist to the in-memory inode. A file can be deleted from disk only when both these counts reach zero.

List xv6 kernel code functions/algorithms for $ rm a – 2M

writei
iupdate
bzero (sys_unlink, iunlockput, iput, itrunc, bfree, bzero)
bfree
iupdate (length; itrunc)
iupdate (type=free; iput)

read system call steps – 2M

ssize_t read(int *fildes*, void **buf*, size_t *nbyte*);
Maurice J. Bach, The Design of The Unix Operating System, 2013 PHI Publishing, section: 5.2, page number: 97
Figure 5.5. Algorithm for Reading a File

Describe filealloc, filedup, and fileclose.- 2M

The functions filealloc, filedup, and fileclose manipulate the global filetable. When an open file is being closed, and its reference count is zero, then the in-memory reference to the inode is also released via iput. Note how the global file table lock is released before calling iput, in order not to hold the lock and perform disk operations.

5. A. In xv6, Explain the working of open("a/b/c", O_RDWR). List xv6 kernel code functions/algorithms used

5A. xv6 functions – 6M

- open("a/b/c", O_RDWR)
  - sys_open
  - namei should return inode for "a/b/c"
    - namei
    - namex
    - start in cp->cwd
    - skipelem: a/b/c -> b/c, name=a
    - look up "a": dirlookup iterates over all entries in dir
    - go back around looking up "b/c" in the "a" dir
    - look up "b", ...
    - return inode for "a/b/c"
  - back in sys_open, get a new FD#
  - this is where we save reference to actual inode
  - return FD# to user

5B.

Explain about structure of a regular file and bmap algorithm in detail. Given a disk-block size of 1 KB and block-pointer address value of 8 bytes, what is the largest file size (in bytes) that can be accessed using 10 direct addresses and one indirect block?

bmap algorithm - 3 M

Maurice J. Bach, The Design of The Unix Operating System, 2013 PHI Publishing, section: 4.2

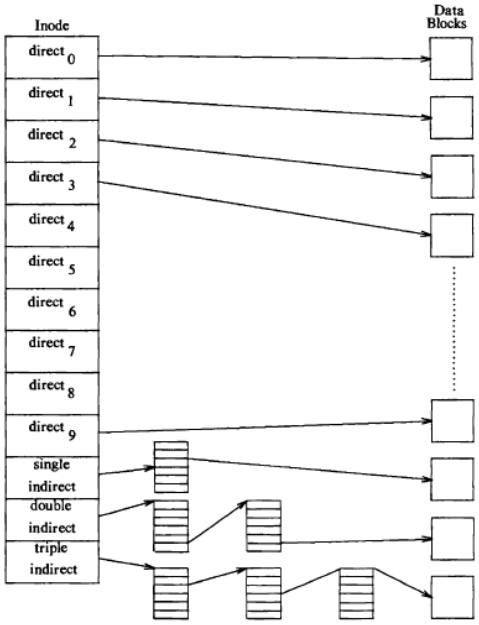


Figure 4.6. Direct and Indirect Blocks in Inode

```
algorithm bmap    /* block map of logical file byte offset to file system block */
input:  (1) inode
        (2) byte offset
output: (1) block number in file system
        (2) byte offset into block
        (3) bytes of I/O in block
        (4) read ahead block number
{
    calculate logical block number in file from byte offset;
    calculate start byte in block for I/O;              /* output 2 */
    calculate number of bytes to copy to user;          /* output 3 */
    check if read-ahead applicable, mark inode;         /* output 4 */
    determine level of indirection;
    while (not at necessary level of indirection)
    {
        calculate index into inode or indirect block from
                logical block number in file;
        get disk block number from inode or indirect block;
        release buffer from previous disk read, if any (algorithm brelse);
        if (no more levels of indirection)
            return (block number);
        read indirect disk block (algorithm bread);
        adjust logical block number in file according to level of indirection;
    }
}
```

Figure 4.8. Conversion of Byte Offset to Block Number in File System

10 direct blocks. One indirect block, which holds 128 blocks. So, the largest file size will be 138blocks which is 138 KB - 3. 5 M

6. **Suppose a process wants to write a few bytes. Let's assume we want to write 100 bytes, starting with byte 2000 in the file. This will be expressed by the pair of system calls: seek(fd,2000);**
   write(fd,buf,100);
   Let's also assume that each disk block is 1024 bytes. Illustrate how Writing may require new blocks to be allocated according to its internal structure, algorithms, and data structures.

6.A.

The data we want to write spans the end of the second block to the beginning of the third block. The problem is that disk accesses are done in fixed blocks, and we only want to write part of such a block. Therefore, the full block must first be read into the buffer cache. Then the part being written is modified by overwriting it with the new data. In our example, this is done with the second block of the file, which happens to be block 8 on the disk. The rest of the data should go into the third block, but the file currently only has two blocks. Therefore, a third block must be allocated from the pool of free blocks. Let's assume that block number 2 on the disk was free, and that this block was allocated. **3M**

As this is a new block, there is no need to read it from the disk before modifying it - we just allocate a block in the buffer cache, prescribe that it now represents block number 2, and copy the requested data to it. Finally, the modified blocks are written back to the disk. Note that the copy of the file's inode was also modified, to reflect the allocation of a new block. Therefore, this too must be copied back to the disk. Likewise, the data structure used to keep track of free blocks needs to be updated on the disk as well. **3.5M**

6.B Explain Inode Life Cycle with ialloc(), iput() algorithms. Typical accessing inode example in Xv6 source code.

```
ip = iget(dev, inum);
ilock(ip); / *... examine and modify ip->xxx ... * /
iunlock(ip);
iput(ip);
```

ialloc() algorithm explanation- 3 M

iput() algorithm explanation - 3 M

Maurice J. Bach, The Design of The Unix Operating System, 2013 PHI Publishing, section: 4.1.2, 4.6

Figure 4.4. Releasing an mode

Figure 4.12. Algorithm for Assigning New Modes

7. **An OS supports a system call sleep, which puts the program making the call to sleep for the number of seconds indicated in the argument of the sleep call. Explain how this system call is implemented.**

sleep algorithm explanation - 4.5 M

xv6's implementation of sleep

```
void sleep(void *chan, struct lock *lk):
if (lk != &ptable.lock) {
acquire(&ptable.lock);
release(lk);
```

```
}
curproc->chan = chan //curproc points to the PCB of current process
curproc->state = SLEEPING
sched() //recall that the scheduler (or the next process)
//releases ptable.lock
```

or

Maurice J. Bach, The Design of The Unix Operating System, 2013 PHI Publishing, section: 6.6
Figure 6.31. Sleep Algorithm

8. **Explain the xv6 code for mycpu and myproc.**
mycpu – 2M
myproc – 2.5 M
Russ Cox, Frans Kaashoek, Robert Morris, xv6: a simple, Unix-like teaching operating system", Revision
https://pdos.csail.mit.edu/6.828/2018/xv6/book-rev11.pdf, Chapter: 5, Page number: 65
xv6 maintains a struct cpu for each processor, which records the process currently running on the processor (if any), the processor's unique hardware identifier (apicid), and some other information. The function **mycpu** returns the current processor's struct cpu. mycpu does this by reading the processor identifier from the local APIC hardware and looking through the array of struct cpu for an entry with that identifier. The return value of mycpu is fragile: if the timer were to interrupt and cause the thread to be moved to a different processor, the return value would no longer be correct. To avoid this problem, xv6 requires that callers of mycpu disable interrupts, and only enable them after they finish using the returned struct cpu.
The function **myproc** returns the struct proc pointer for the process that is running on the current processor. myproc disables interrupts, invokes mycpu, fetches the current process pointer (c->proc) out of the struct cpu, and then enables interrupts. If there is no process running, because the the caller is executing in scheduler, myproc returns zero. The return value of myproc is safe to use even if interrupts are enabled: if a timer interrupt moves the calling process to a different processor, its struct proc pointer will stay the same.

9. **(i). How many levels does the UNIX scheduling algorithm include? What are they? In its low-level algorithm, how is the priority value for every process computed? What does each of the three components in the priority formula mean, respectively? How does the priority formula indicate that UNIX gives higher priority to processes that have used less CPU time in the recent past? Explain the reason by describing the computation process of the priority formula. 4M**
There are two levels of unix scheduling algorithm. They are user level and kernel level.
Processes that sleep in lower-level algorithms tends to cause more system bottlenecks the longer they are inactive; hence they receive a higher priority than processes that would cause fewer system bottlenecks.
priority--- ("recent CPU usage"/2) + (base level user priority)
In this formula, "base level user priority" is the threshold priority between kernel and user mode. When it recomputes recent CPU usage, the clock handler also recalculates the priority of every process in the "preempted but ready-to-run" state according to the above formula.
A numerically low value implies a high scheduling priority. Examining the functions for recomputation of recent CPU usage and process priority, the slower the decay rate for recent CPU usage, the longer it will take for the priority of a process to reach its base level; consequently, processes in the "ready-to-run" state will tend to occupy more priority levels.
Kernel assigns priority to a process about to go to sleep, correlating a fixed, priority value with the reason for sleeping. The priority does not depend on the runtime characteristics of the process (I/O bound or CPU bound), but instead is a constant value that is hard-coded for each call to sleep, dependent on the reason the process is sleeping.
Processes can exercise crude control of their scheduling priority by using the nice
system call: nice (value);
where value is added in the calculation of process priority:
priority= ("recent CPU usage"/constant) + (base priority) + (nice value)
Maurice J. Bach, The Design of The Unix Operating System, 2013 PHI Publishing, section: 8.1.2
Figure 8.3. Movement of a Process on Priority Queues
(ii). Write a system program to execute a command and redirect the output to a file : $ wc sample.txt > newfile. 4 M

```
/* Program: dup2.c -- Opens files in the parent and uses dup2 in the child to reassign the stdout descriptor */
#include <unistd.h>
#include <stdio.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <wait.h>
#define OPENFLAGS (O_WRONLY | O_CREAT | O_TRUNC)
#define MODE6OO (S_IRUSR | S_IWUSR)
int main(int argc, char **argv) {
int fd2, rv, exit_status;
if (fork() == 0) { /* Child */
perror("Error in opening file for reading\n");
exit(1); }
if ((fd2 = open(argv[1], OPENFLAGS, MODE6OO)) == -1){
perror("Error in opening file for writing\n");
```

```
exit(2); }
dup2(fd2,1); /* Closes standard output simultaneously */
execvp(argv[2], &argv[2]); /* Execute command */
perror("exec error");
exit(3);
} else { /* Parent */
wait(&rv); /* Or use waitpid(-1, &rv, 0) */
printf("Exit status: %d\n", WEXITSTATUS(rv));
}
}
```

10. (i). In xv6, explain the purpose of init.c. How shell works give an algorithm? 4 M

init.c – 2 M

1.The PPID of every login shell is always 1. This is the init process: the second process of the system.
2. init is a very important process and, apart from being the parent of users' shells, it is also responsible for giving birth to every service that's running in the system—like printing, mail, Web, and so on.
3. Even though no one may be using the system, a number of system processes keep running all the time.
4. They are spawned during system startup by init (PID 1), the parent of the login shell. The ps -e command lists them all.
5. System processes that have no controlling terminal are easily identified by the ? in the TTY column.
6. A process that is disassociated from the terminal can neither write to the terminal nor read from it. Such processes are also known as daemons. Many of these daemons are actually sleeping (a process state) and wake up only when they receive input.
7. Examples of Daemons are: lpsched, sendmail, inetd etc.
8. To initialize a system from an inactive state, an administrator goes through a "bootstrap" sequence: The administrator "boots" the system. Boot procedures vary according to machine type, but the goal is common to all machines.
9. The init process is a process dispatcher, spawning processes that allow users to log in to the system, among others.
10. Init reads the file "tetchnittab" for instructions about which processes to spawn.
11. The file "/etc/inittab" contains lines that contain an "id," a state identifier (single user, multi-user, etc.), an "action" and a program specification.
12. Init reads the file and, if the state in which it was invoked matches the state identifier of a line, creates a process that executes the given program specification

How shell works- 2M

```
while (TRUE)
{
  Display the shell prompt on the screen;
  Read the input line from keyboard;
  pid = fork();                    /* create a child process */
  if (pid<0)
  {
   Display error message on the screen;
    continue;
   }
   if (pid != 0)
    {                               /* code for the parent process*/
     waitpid (-1,&staus, 0);        /* the parent process waits the child process */
    }
  else
  {                                 /* code for child process*/
    execve (command, parameters, 0);   /* execute the command */
  }
}
```

(ii). Explain algorithms for stime, time, times, and clock.Init.c 4 M

There are several time-related system calls, stime, time, times, and alarm. The first two deal with global system time, and the latter two deal with time for individual processes. Stime allows the superuser to set a global kernel variable to a value that gives the current time: stime (pvalue); where pvaluepoints to a long integer that gives the time as measured in seconds from midnight before (00:00:00) January 1, 1970, GMT. The clock interrupt handler increments the kernel variable once a second. Time retrieves the time as set by stime: time(tloc); where tlocpoints to a location in the user process for the return value.

Maurice J. Bach, The Design of The Unix Operating System, 2013 PHI Publishing, section: 8.3, Page Number: 261

Figure 8.9. Algorithm for the Clock Handler

11. A.

List the various sections of the disk image of an executable file in UNIX. 2M

UNIX supports many executable file formats. The oldest is the a.out format, which has a 32-byte header followed by text and data sections and the symbol table. The program header contains the sizes of the text, initialized data, and uninitialized data regions, and the entry point, which is the address of the first instruction the program must execute. It also contains a magic number, which identifies the file as a valid executable file and gives further information about its format, such as whether the file is demand paged, or whether the data section begins on a page boundary. Each UNIX variant defines the set of magic numbers it supports.

In xv6, explain the algorithm for a system call that makes a process to overwrite itself with another executable image. 4M

Exec system call that makes a process to overwrite itself with another executable image.
The exec system call must perform the following tasks:
1. Parse the pathname and access the executable file.
2. Verify that the caller has execute permission for the file.
3. Read the header and check that it is a valid executable.
4. If the file has SUID or SGID bits set in its mode, change the caller's effective UID or GID respectively to that of the owner of the file.
5. Copy the arguments to exec and the environment variables into kernel space, since the current user space is going to be destroyed.
6. Allocate swap space for the data and stack regions.
7. Free the old address space and the associated swap space. If the process was created by vfork, return the old address space to the parent instead.
8. Allocate address maps for the new text, data, and stack.
9. Set up the new address space. If the text region is already active (some other process is already running the same program}, share it with this process. Otherwise, it must be initialized from the executable file. UNIX processes are usually demand paged, meaning that each page is read into memory only when the program needs it.
10. Copy the arguments and environment variables back onto the new user stack.
11. Reset all signal handlers to default actions, because the handler functions do not exist in the new program. Signals that were ignored or blocked before calling exec remain ignored or blocked.
12. Initialize the hardware context. Most registers are reset to zero, and the program counter is set to the entry point of the program.

11. B

The traditional UNIX scheduler is a priority-based round robin scheduler (also called a multi-level round robin scheduler). How does the scheduler go about favoring I/O bound jobs over long-running CPU-bound jobs? 2M

It uses multiple ready queues of each priority. priorities are gradually increased over time to prevent starvation of low priority processes. They are increased by boosting the priority based on the amount of CPU consumed. I/O bound jobs are indirectly favoured as they consume less CPU.

For the given list of processes and service time: P1 120, P2 60, P3 180, p4 50, P5 300 Answer the following: 4.5 M

(i). Draw a Gantt chart that shows the completion times for each process using first-come, first served CPU scheduling.
(ii). Draw a Gantt chart that shows the completion times for each process using shortest-job-next CPU scheduling.
(iii). Draw a Gantt chart that shows the completion times for each process using round-robin CPU scheduling with a time slice of 60.

Process	P1	P2	P3	P4	P5
Service time	120	60	180	50	300

Draw a Gantt chart that shows the completion times for each process using first-come, first served CPU scheduling.

```
0          120    180             360 410                710
 P1          P2     P3               P4  P5
```

Draw a Gantt chart that shows the completion times for each process using shortest-job-next CPU scheduling.

```
 0    50     110          230      410                  710
 P4    P2     P1           P3       P5
```

Draw a Gantt chart that shows the completion times for each process using round-robin CPU scheduling with a time slice of 60.

Process	P1	P2	P3	P4	P5	P1	P3	P5	P3	P5	P5	P5
time	60	60	60	50	60	60	60	60	60	60	60	60

12. A. Show a pictorial arrangement - Sharing of kernel data structures and open files between parent and child after fork. 3.5 M

The logical view of the parent and child processes and their relationship to other kernel data structures immediately after completion of the fork system call. To summarize, both processes share files that the parent had open at the time of the fork, and the file table reference count for those files is one greater than it had been. Similarly, the child process has the same current directory and changed root (if applicable) as the parent, and the mode reference count of those directories is one greater than it had been. The processes have identical copies of the text, data, and (user) stack regions; the region type and the system implementation determine whether the processes can share a physical copy of the text region.

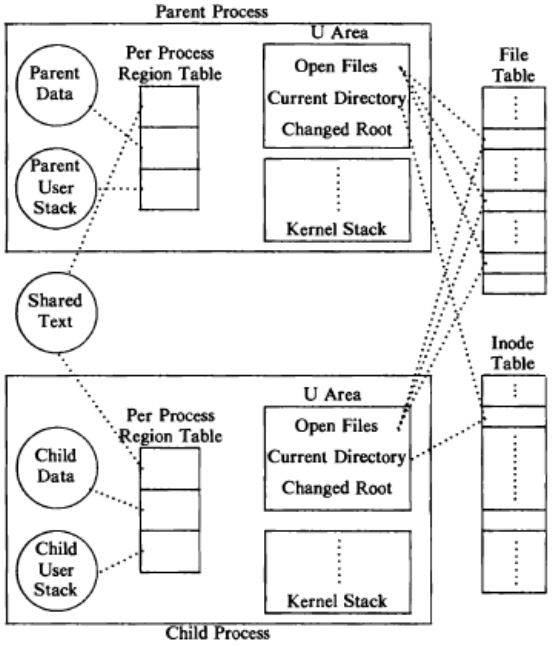


Figure 7.3. Fork Creating a New Process Context

Maurice J. Bach, The Design of The Unix Operating System, 2013 PHI Publishing, section: 7.1
Figure 7.3. Fork Creating a New Process Context

What elements of the process context must the kernel explicitly save when handling (i).context switch, (ii) an interrupt, or (iii) a system call? What are the similarities and differences? 3 M

The context of a process consists of the contents of its (user) address space and the contents of hardware registers and kernel data structures that relate to the process. Formally, the context of a process is the union of its user-level context, register context, and system-level context! The user-level context consists of the process text, data, user stack, and shared memory that occupy the virtual address space of the process.

Maurice J. Bach, The Design of The Unix Operating System, 2013 PHI Publishing, section: 6.3
Figure 6.8. Components of the Context of a Process

There are 4 situations under which kernel permits a context switch are :

- When a process is in sleep mode
- When a process goes to exit
- When a process returns from a system call to user mode but is not the most eligible process to run
- When a process returns from an interrupt handler to user mode but is not the most eligible process to run

A running process will always result in a context switch of the running process, even in a non preemptive kernel design when

a) A blocking system call.
b) The system call exit, to terminate the current process

After finishing its job in kernel mode, the OS may sometimes decide to go back to the user mode of the same process, without switching to another process.

Sequence of events happen during a context switch from (user mode of) process P to (user mode of) process Q, triggered by a timer interrupt that occurred when P was executing, in a Unix-like operating system design

a) The CPU executing process P moves from user mode to kernel mode.
b) The OS scheduler code is invoked.
c) The CPU stack pointer moves from the kernel stack of P to the kernel stack of Q.
d) The CPU program counter moves from the kernel address space of P to the kernel address space of Q.
e) The CPU program counter moves from the kernel address space of Q to the user address space of Q.

12. B. What do we mean by race condition in the context of multiple processes? Explain race for locked Buffer and free Buffer with a diagram. 4-M

A race condition occurs when two or more processes can access shared data and they try to change it at the same time.

Maurice J. Bach, The Design of The Unix Operating System, 2013 PHI Publishing, section: 3.3, Page numbers: 51, 53
Figure 3.10. Race for Free Buffer
Figure 3.12. Race for a Locked Buffer

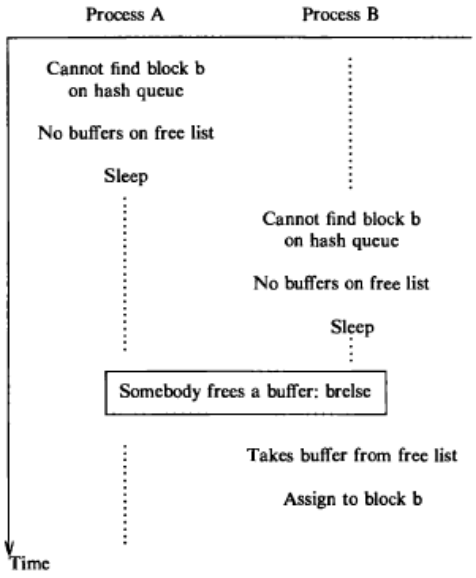


Figure 3.10. Race for Free Buffer

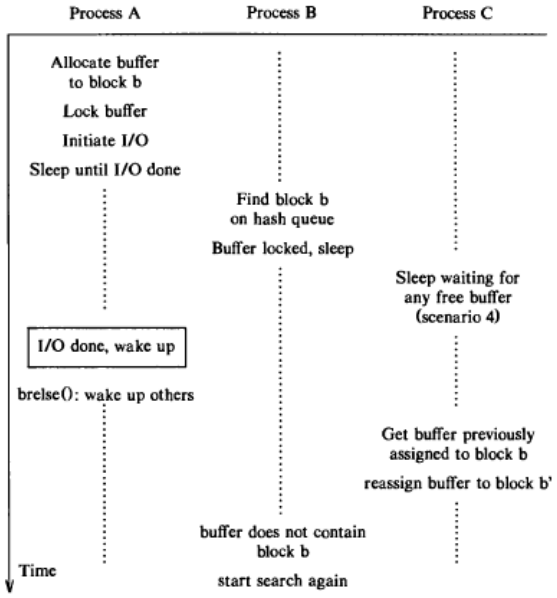


Figure 3.12. Race for a Locked Buffer

Give solution using locks. Illustrate xv6 spinlocks. 2 M

xv6 runs on a multiprocessor and allows multiple CPUs to execute concurrently inside the kernel. These usually correspond to system calls made by processes running on different CPUs. There might also be interrupt handlers running at the same time as other kernel code. An interrupt can occur on any of the CPUs; both user-level processes and processes inside the kernel can be interrupted. xv6 uses spin-locks to coordinate how these concurrent activities share data structures.

Spinning vs blocking

Spin-locks are good when protecting short operations: increment a counter, search in i-node cache, allocate a free buffer. In these cases acquire() won't waste much CPU time spinning even if another CPU holds the lock.

Spin locks are not so great for code that has to wait for events that might take a long time. For example, it would not be good for the disk reading code to get a spin-lock on the disk hardware, start a read, wait for the disk to finish reading the data, and then release the lock. The disk often takes 10s of milliseconds (millions of instructions) to complete an operation. Spinning wastes the CPU; if another process is waiting to execute, it would be better to run that process until the disk signals it is finished by causing an interrupt.

xv6 has sleep() and wakeup() primitives that are better for processes that need to wait for I/O.