Chapter 14 and 15

Program Swapping &

Introduction to Basic I/0

- Rupesh Jain
- Mikel Rodriguez

Outline

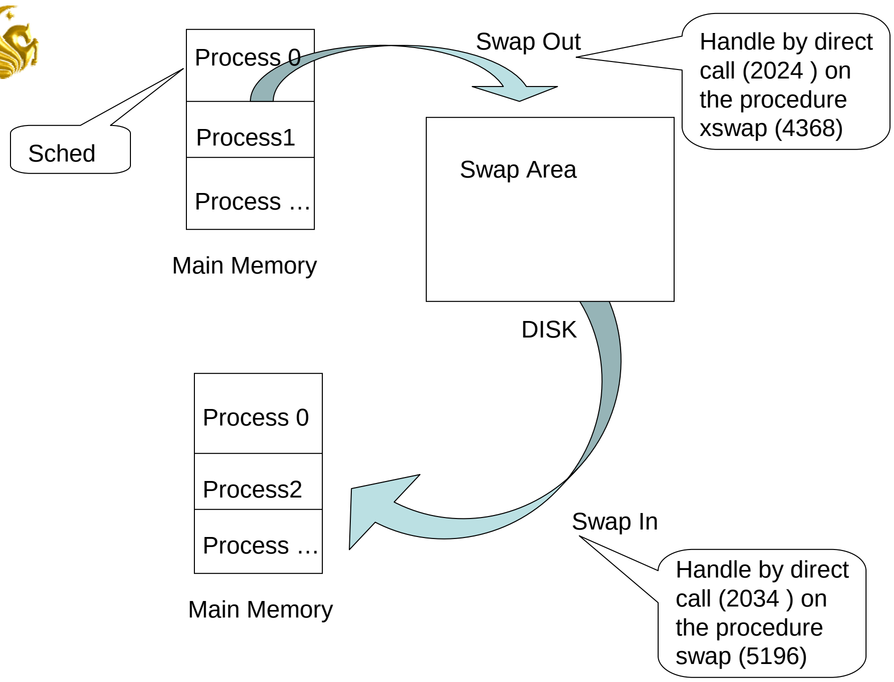
1. Introduction
2. Understanding the Sched Procedure
3. Walkthrough of the program swapping code
4. Four procedure manipulating the text array
5. Basic I/O under Unix
6. The file “Buff.h”
7. The file “Conf.c”
8. The Swap procedure
9. Race Conditions
10. Summary
11. References

What is Program Swapping ?

Processes can be selectively swapped out and swap in to share the limited resource of main physical memory among several process

It is also called roll in and roll out

UNIX like other timesharing and multiprogramming system uses Program swapping



History

Originally Sched was called “swap” directly to swap out rather than xswap

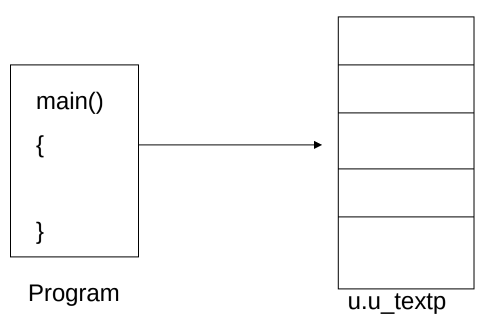
Four Procedure which manipulate the array of structures called text

1. xswap
2. xalloc
3. xfree
4. xccdec

Text Segment

Text segment are segment which contain only pure code and data which remains unaltered throughout the program execution

Information about text segment must be stored in a central location (i.e text array)



Sched (1940)

Sched spends most of its time waiting in one of the following situation

A. (runout)

- None of the processes which are swapped out is ready to run
- The situation can be changed by a call to “wakeup” or to “xswap” called by “newproc”.

B. (runin)

- There is atleast one process swapped out and ready to run but it hasn’t been out more than 3 seconds and none of the process present in main memory is inactive or has been more than 2 seconds.
- The situation can be changed by a call to “sleep”

Sched Procedure

```
1940 sched()
1941 {
1942     struct proc *p1;
1943     register struct proc *rp;
1944     register a, n;
1951     goto loop;
1957 loop:
1958     spl6();
1959     n = -1;
1960     for(rp = &proc[0]; rp < &proc[NPROC]; rp++)
1961     if(rp->p_stat==SRUN && (rp->p_flag&SLOAD)==0 &&
1962         rp->p_time > n) {
1963             p1 = rp;
1964             n = rp->p_time;
1965     }
```

Find the process ready to run,select the longest one

A search made for the process which is ready to run and has been swapped out for the longest time

```
1966    if(n == -1) {
1967            runout++;
1968            sleep(&runout, PSWP);
1969            goto loop;
1970    }
1971
1972    /*
1973     * see if there is core for that process
1974     */
1975
1976    spl0();
1977    rp = p1;
1978    a = rp->p_size;
1979    if((rp=rp->p_textp) != NULL)
1980            if(rp->x_ccount == 0)
1981                    a =+ rp->x_size;
1982    if((a=malloc(coremap, a)) != NULL)
1983            goto found2;
1984
1985    /*
1986     * none found,
1987     * look around for easy core
1988     */
1989
1990    slp6();
1991    for(rp = &proc[0]; rp < &proc[NPROC]; rp++)
1992            if((rp->p_flag&(SSYS|SLOCK|SLOAD))==SLOAD &&
1993                (rp->p_stat == SWAIT || rp->p_stat==SSTOP))
1994                    goto found1;
```

There is no such process, situation A holds

main memory area for to hold data segment if text segment needs to be present also text segment the size needs to be increased

If it has adequate size text and data then goto found2(2031)

If the process is waiting for event of low precedence &which is not locked and state is swait or sstop but not ssleep then goto swap out

```
2003    if(n < 3)
2004            goto sloop;
2005    n = -1;
2006    for(rp = &proc[0]; rp < &proc[NPROC]; rp++)
2007            if((rp->p_flag&(SSYS|SLOCK|SLOAD))==SLOAD &&
2008                (rp->p_stat==SRUN || rp->p_stat==SSLEEP) &&
2009                 rp->p_time > n) {
2010                    p1 = rp;
2011                    n = rp->p_time;
2012    }
2013    if(n < 2)
2014            goto sloop;
2015    rp = p1;
2016
2017    /*
2018     * swap user out
2019     */
2020
2021 found1:
2022    slp0();
2023    rp->p_flag =& ~SLOAD;
2024    xswap(rp, 1, 0);
2025    goto loop;
2026
2027    /*
2028     * swap user in
2029     */
2030
2031 found2:
2032    if((rp=p1->p_textp) != NULL) {
2033            if(rp->x_ccount == 0) {
2034                    if(swap(rp->x_daddr, a, rp->x_size, B_READ))
2035                            goto swaper;
2036                    rp->x_caddr = a;
2037                    a =+ rp->x_size;
2038            }
2039            rp->x_ccount++;
2040    }
2041    rp = p1;
2042    if(swap(rp->p_addr, a, rp->p_size, B_READ))
2043            goto swaper;
2044    mfree(swapmap, (rp->p_size+7)/8, rp->p_addr);
2045    rp->p_addr = a;
2046    rp->p_flag =| SLOAD;
2047    rp->p_time = 0;
2048    goto loop;
2049
```

If image to be swapped in has been <3 sec ,B holds

Search for the process which is loaded but not locked whose state is srun or ssleep (waiting for high precedence) and been in mem for longest time

Process swapped out is <2 sec ,situation B holds

Swap out using xswap and process image is flagged as not loaded

Read the text seg into mm

Swap(addr within swap area, mm addr, size, direction indicator)

Release disk swap area to the available list,record mm addr,set the sload reset the accumulated time indicator

xswap(4368)

```
4368 xswap(p, ff, os)
4369 int *p;
4370 {   register *rp, a;
4371
4372     rp = p;
4373     if(os == 0)
4374             os = rp->p_size;
4375     a = malloc(swapmap, (rp->p_size+7)/8);
4376     if(a == NULL)
4377             panic("out of swap space");
4378     xccdec(rp->p_textp);
4379     rp->p_flag =| SLOCK;
4380     if(swap(a, rp->p_addr, os, 0))
4381             panic("swap error");
4382     if(ff)
4383             mfree(coremap, os, rp->p_addr);
4384     rp->p_addr = a;
4385     rp->p_flag =& ~(SLOAD|SLOCK);
4386     rp->p_time = 0;
4387     if(runout) {
4388             runout = 0;
4389             wakeup(&runout);
4390     }
```

If oldsize was not supplied then use current size of data segment

Find the disk space area for the process data segments

xccdec is called unconditionally to decrement the count associated with the text segment of the number of "in mm"processes which reference that text segment. If the count becomes zero, the mm area occupied by the text seg is simply returned to the available space.

The Slockflag is set while the process is being swapped out

mm image is released except when xswap is called by newproc

runout is set, sched is waiting for something to swap in so wake it up

xalloc (4433)

It is called by "exec"(3130) when new program is being initiated.

Its handle the allocation of ,or linking to ,the text segment.

xalloc (ip) ,the argument ip is a pointer to the mode of the code file.

xfree (4398)

xfree is called by "exit" (3233) when a process is being terminated

```
4401     if((xp=u.u_procp->p_textp) != NULL) {
4402             u.u_procp->p_textp == NULL;
4403             xccdec(xp);
4404             if(--xp->x_count == 0) {
4405                     ip = xp->x_iptr;
4406                     if((ip->i_mode&ISVTX) == 0) {
4407                             xp->x_iptr = NULL;
4408                             mfree(swapmap, (xp->x_size+7)/8,
4409                                     xp->x_daddr);
4410                             ip->i_flag =& ~ITEXT;
4411                             iput(ip);
4412                     }
4413             }
```

Set the text pointer in the proc entry to Null

Decrement mem count

Text segment is not flagged to be saved

abandon the text segment in the disk swap area

An Introduction to

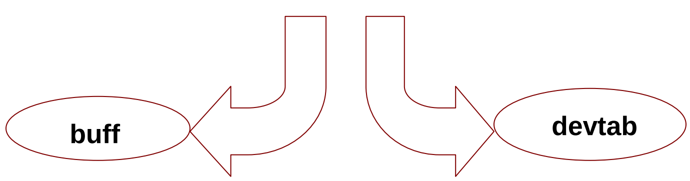
Basic I/O under Unix

There are three files whose contents are essential to UNIX input/output:

- “buf.h”
- “conf.h”
- “conf.c”

The File “buf.h”

This file declares two structures:



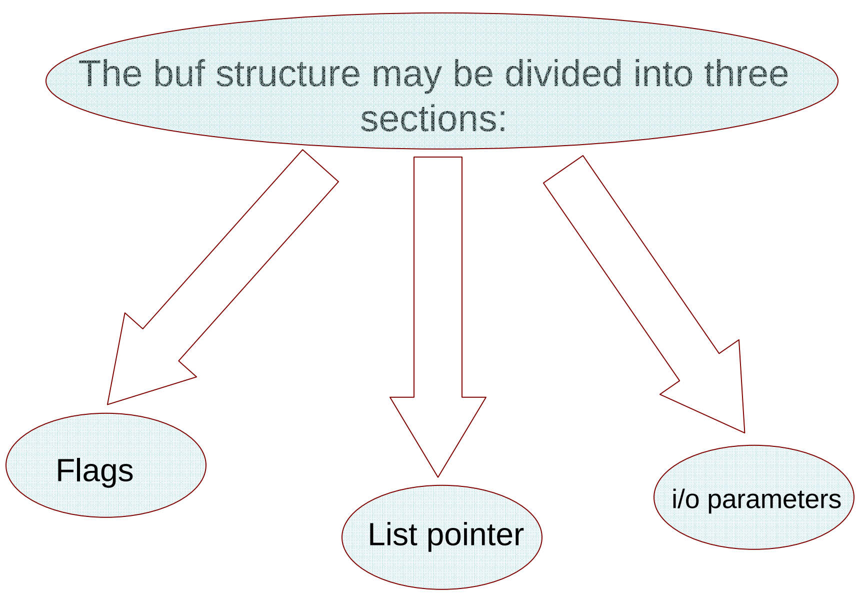
"struct buf"

This structure is buffer header and serves as a buffer control block

```
struct buf
{
 int      b_flags;
 struct   buf *b_forw;
 struct   buf *b_back;
 struct   buf *av_forw;
 struct   buf *av_back;
 int      b_dev;
 int      b_wcount;
 char     *b_addr;
 char     *b_xmem;
 char     *b_blkno;
 char     b_error;
 char     *b_resid;
```

- b_flags → Status flags
- av_forw, av_back → Position on free list
- b_dev → Major+minor device name
- b_wcount → transfer count
- b_blkno → block # on device
- b_error → returned after I/O
- b_resid → words not transferred

"struct buf" (continued)



“devtab struct”

The devtab structure contains status information for the devices and serves as a list head for:

(a) the list of buffers associated with the device

(b) the list of outstanding i/o requests for the device

The File “conf.h”

The file “conf.h” declares:

```
struct      {
            char     d_minor;
            char     d_major;
};
```

```
struct      bdevsw  {
   int      (*d_open)();
   int      (*d_close)();
   int      (*d_strategy)();
   int      *d_tab;
} bdevsw[];
```

```
int         nblkdev;
```

The File “conf.c”

```
int (*bdevsw[])()
{
 &nulldev, &nulldev, &rkstrategy, &rktab, /*rk */
 &nodev, &nodev, &nodev, 0, /* rp */
 &nodev, &nodev, &nodev, 0, /* rf */
 &nodev, &nodev, &nodev, 0, /* tm */
 &nodev, &nodev, &nodev, 0, /* tc */
 &nodev, &nodev, &nodev, 0, /* hs */
 &nodev, &nodev, &nodev, 0, /* hp */
 &nodev, &nodev, &nodev, 0, /* ht */
 0
};
```

```
int (*cdevsw[])()
{
 &klopen, &klclose, &klread, &klwrite, &klsgtty,
                                     /* console */
 &pcopen, &pcclose, &pcread, &pcwrite, &nodev,
                                     /* pc */
 &lpopen, &lpclose, &nodev, &lpwrite, &nodev,
                                     /* lp */
 &nodev, &nodev, &nodev,  &nodev,  &nodev, /* dc */
 &nodev, &nodev, &nodev,  &nodev,  &nodev, /* dh */
 &nodev, &nodev, &nodev,  &nodev,  &nodev, /* dp */
 &nodev, &nodev, &nodev,  &nodev,  &nodev, /* dj */
 &nodev, &nodev, &nodev,  &nodev,  &nodev, /* dn */
 &nulldev, &nulldev, &mmread, &mmwrite, &nodev,
                                     /* mem */
 &nulldev, &nulldev, &rkread, &rkwrite, &nodev,
                                     /* rk */
 &nodev, &nodev, &nodev,  &nodev,  &nodev, /* rf */
 &nodev, &nodev, &nodev,  &nodev,  &nodev, /* rp */
 &nodev, &nodev, &nodev,  &nodev,  &nodev, /* tm */
 &nodev, &nodev, &nodev,  &nodev,  &nodev, /* hs */
 &nodev, &nodev, &nodev,  &nodev,  &nodev, /* hp */
 &nodev, &nodev, &nodev,  &nodev,  &nodev, /* ht */
 0
};
```

Swbuf in “bio.c”

“Swbuf” controls swapping input and output

```
        fp = &swbuf.b_flags;
        spl6();
        while (*fp&B_BUSY) {
                *fp =| B_WANTED;
                sleep(fp, PSWP);
        }
        *fp = B_BUSY | B_PHYS | rdflg;
        swbuf.b_dev = swapdev;
        swbuf.b_wcount = - (count<<5);  /* 32 w/block */
        swbuf.b_blkno = blkno;
        swbuf.b_addr = coreaddr<<6;     /* 64 b/block */
        swbuf.b_xmem = (coreaddr>>10) & 077;
        (*bdevsw[swapdev>>8].d_strategy)(&swbuf);
        spl6();
        while((*fp&B_DONE)==0)
                sleep(fp, PSWP);
        if (*fp&B_WANTED)
                wakeup(fp);
        spl0();
        *fp =& ~(B_BUSY|B_WANTED);
        return(*fp&B_ERROR);
}
```

Swbuf Continued (An Example)

The code for swap displays some of the problems of race conditions when several processes are running together

Process "A" initiates a swapping operation

A

A

A

```
fp = &swbuf.b_flags;
spl6();
while (*fp&B_BUSY) {
        *fp =| B_WANTED;
        sleep(fp, PSWP);
}
*fp = B_BUSY | B_PHYS | rdflg;
swbuf.b_dev = swapdev;
swbuf.b_wcount = - (count<<5);  /* 32 w/block */
swbuf.b_blkno = blkno;
swbuf.b_addr = coreaddr<<6;     /* 64 b/block */
swbuf.b_xmem = (coreaddr>>10) & 077;
(*bdevsw[swapdev>>8].d_strategy)(&swbuf);
spl6();
while((*fp&B_DONE)==0)
        sleep(fp, PSWP);
if (*fp&B_WANTED)
        wakeup(fp);
spl0();
*fp =& ~(B_BUSY|B_WANTED);
return(*fp&B_ERROR);
}
```

Flags==B_BUSY | B_PHYS | rdflg

Swbuf Continued (An Example)

Process "B" initiates a swapping operation

Flags==B_BUSY | B_PHYS | rdflg

B

B

A

```
fp = &swbuf.b_flags;
spl6();
while (*fp&B_BUSY) {
        *fp =| B_WANTED;
        sleep(fp, PSWP);
}
*fp = B_BUSY | B_PHYS | rdflg;
swbuf.b_dev = swapdev;
swbuf.b_wcount = - (count<<5);  /* 32 w/block */
swbuf.b_blkno = blkno;
swbuf.b_addr = coreaddr<<6;     /* 64 b/block */
swbuf.b_xmem = (coreaddr>>10) & 077;
(*bdevsw[swapdev>>8].d_strategy)(&swbuf);
spl6();
while((*fp&B_DONE)==0)
        sleep(fp, PSWP);
if (*fp&B_WANTED)
        wakeup(fp);
spl0();
*fp =& ~(B_BUSY|B_WANTED);
return(*fp&B_ERROR);
}
```

Swbuf Continued (An Example)

Flags==B_BUSY| B_PHYS | rdflg |B_WANTED

B

```
fp = &swbuf.b_flags;
spl6();
while (*fp&B_BUSY) {
        *fp =| B_WANTED;
        sleep(fp, PSWP);
}
*fp = B_BUSY | B_PHYS | rdflg;
swbuf.b_dev = swapdev;
swbuf.b_wcount = - (count<<5);  /* 32 w/block */
swbuf.b_blkno = blkno;
swbuf.b_addr = coreaddr<<6;     /* 64 b/block */
swbuf.b_xmem = (coreaddr>>10) & 077;
(*bdevsw[swapdev>>8].d_strategy)(&swbuf);
spl6();
while((*fp&B_DONE)==0)
        sleep(fp, PSWP);
if (*fp&B_WANTED)
        wakeup(fp);
spl0();
*fp =& ~(B_BUSY|B_WANTED);
return(*fp&B_ERROR);
}
```

A

Swbuf Continued (An Example)

Flags==B_BUSY| B_PHYS | rdflg |B_WANTED

B

```
fp = &swbuf.b_flags;
spl6();
while (*fp&B_BUSY) {
        *fp =| B_WANTED;
        sleep(fp, PSWP);
}
*fp = B_BUSY | B_PHYS | rdflg;
swbuf.b_dev = swapdev;
swbuf.b_wcount = - (count<<5);  /* 32 w/block */
swbuf.b_blkno = blkno;
swbuf.b_addr = coreaddr<<6;     /* 64 b/block */
swbuf.b_xmem = (coreaddr>>10) & 077;
(*bdevsw[swapdev>>8].d_strategy)(&swbuf);
spl6();
while((*fp&B_DONE)==0)
        sleep(fp, PSWP);
if (*fp&B_WANTED)
        wakeup(fp);
spl0();
*fp =& ~(B_BUSY|B_WANTED);
return(*fp&B_ERROR);
}
```

C

Interrupt

A

FINISHED

Swbuf Continued (An Example)

Flags== B_BUSY | B_PHYS | rdflg | B_DONE

Sets Flags to

C

Wake up

Wake up

A

B

What happens next depends on the order in which A and B are reactivated

```
    fp = &swbuf.b_flags;
    spl6();
    while (*fp&B_BUSY) {
            *fp =| B_WANTED;
            sleep(fp, PSWP);
    }
    *fp = B_BUSY | B_PHYS | rdflg;
    swbuf.b_dev = swapdev;
    swbuf.b_wcount = - (count<<5);  /* 32 w/block */
    swbuf.b_blkno = blkno;
    swbuf.b_addr = coreaddr<<6;     /* 64 b/block */
    swbuf.b_xmem = (coreaddr>>10) & 077;
    (*bdevsw[swapdev>>8].d_strategy)(&swbuf);
    spl6();
    while((*fp&B_DONE)==0)
            sleep(fp, PSWP);
    if (*fp&B_WANTED)
            wakeup(fp);
    spl0();
    *fp =& ~(B_BUSY|B_WANTED);
    return(*fp&B_ERROR);
}
```

Swbuf Continued (An Example)

Case 1: A goes first:

Process B can now finish

B

```
    fp = &swbuf.b_flags;
    spl6();
    while (*fp&B_BUSY) {
            *fp =| B_WANTED;
            sleep(fp, PSWP);
    }
    *fp = B_BUSY | B_PHYS | rdflg;
    swbuf.b_dev = swapdev;
    swbuf.b_wcount = - (count<<5);  /* 32 w/block */
    swbuf.b_blkno = blkno;
    swbuf.b_addr = coreaddr<<6;     /* 64 b/block */
    swbuf.b_xmem = (coreaddr>>10) & 077;
    (*bdevsw[swapdev>>8].d_strategy)(&swbuf);
    spl6();
    while((*fp&B_DONE)==0)
            sleep(fp, PSWP);
    if (*fp&B_WANTED)
            wakeup(fp);
    spl0();
    *fp =& ~(B_BUSY|B_WANTED);
    return(*fp&B_ERROR);
}
```

A

- "B_DONE" is set (so no more sleeping is needed)

A

- "B_WANTED" is reset (so there is no wakeup)

A

Process "A" finishes up and sets: Flags==B_PHYS | rdflg | B_DONE

Swbuf Continued (An Example)

•Process "B" wakes up and finishes

Case 2: B goes first:

Goes to sleep again leaving: Flags==B_BUSY | B_PHYS | rdflg | B_DONE | B_WANTED

```
    fp = &swbuf.b_flags;
    spl6();
    while (*fp&B_BUSY) {
        *fp =| B_WANTED;
        sleep(fp, PSWP);
    }
    *fp = B_BUSY | B_PHYS | rdflg;
    swbuf.b_dev = swapdev;
    swbuf.b_wcount = - (count<<5);  /* 32 w/block */
    swbuf.b_blkno = blkno;
    swbuf.b_addr = coreaddr<<6;     /* 64 b/block */
    swbuf.b_xmem = (coreaddr>>10) & 077;
    (*bdevsw[swapdev>>8].d_strategy)(&swbuf);
    spl6();
    while((*fp&B_DONE)==0)
        sleep(fp, PSWP);
    if (*fp&B_WANTED)
        wakeup(fp);
    spl0();
    *fp =& ~(B_BUSY|B_WANTED);
    return(*fp&B_ERROR);
}
```

•Finds "B_BUSY"

•Turns "B_WANTED" on

Process "A" starts again: it finds B_WANTED on

•Process A calls "wakeup"

Summary

• "Sched" procedure makes the decision of swap in and swap out

• Four procedures manipulate the text array structure
1) xswap
2) xccdec
3) xalloc
4) xfree

•The "buff" structure in "buf.h" is buffer header and serves as a buffer control block.
•"Swbuf" (an instance of the buf structure) controls the process of swapping input and output.

References

- ***"Lions' Commentary on Unix 6th edition with source code"*** **by John Lion.**
- **http://www.uwsg.iu.edu/UAU/memory/pageswap.html** ***"Unix For Advanced Users"***
- ***" The Unix I/O System"*** **by Dennis Ritchie**
- ***"UNIX in a Nutshell, 3rd Edition"*** **by Arnold Robbins**