Initialization and Process Initiation in UNIX

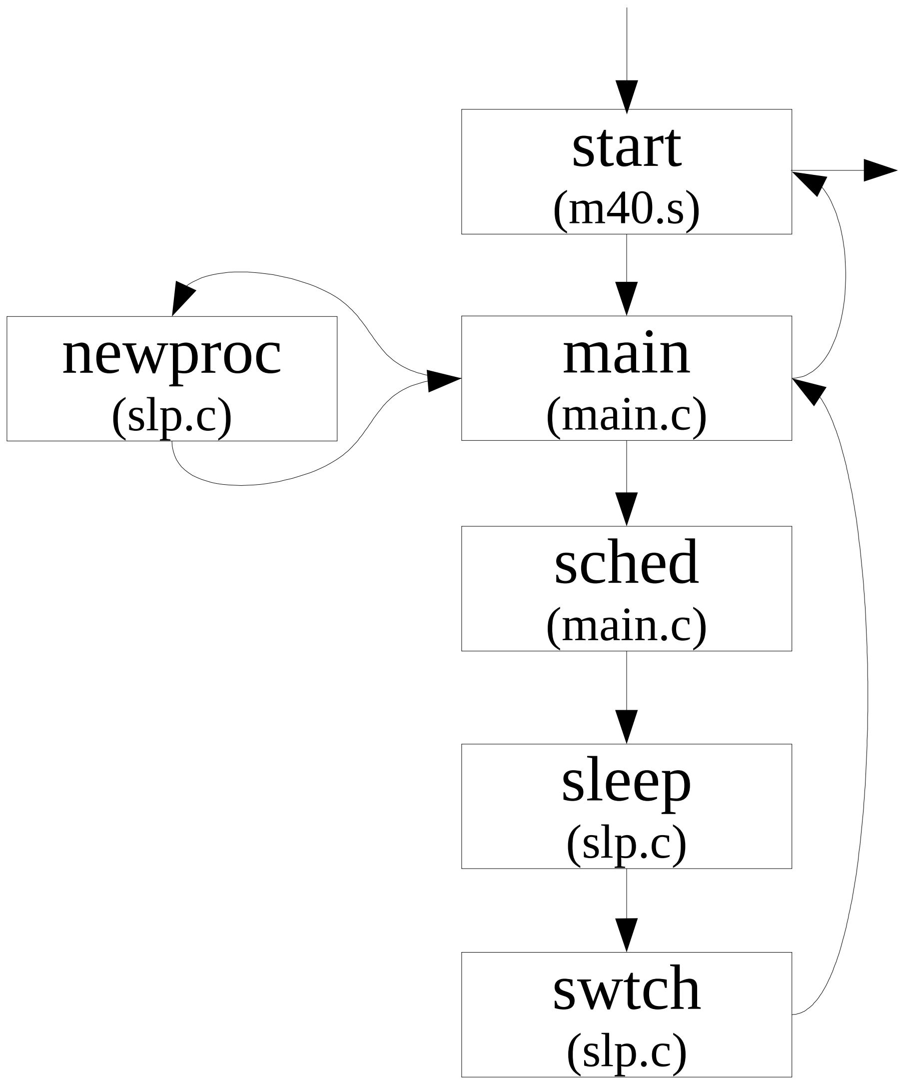
Chapters 6 and 7 from Lions' Commentary on UNIX

Presented by Wade Spires
COP 5611
March 22, 2005

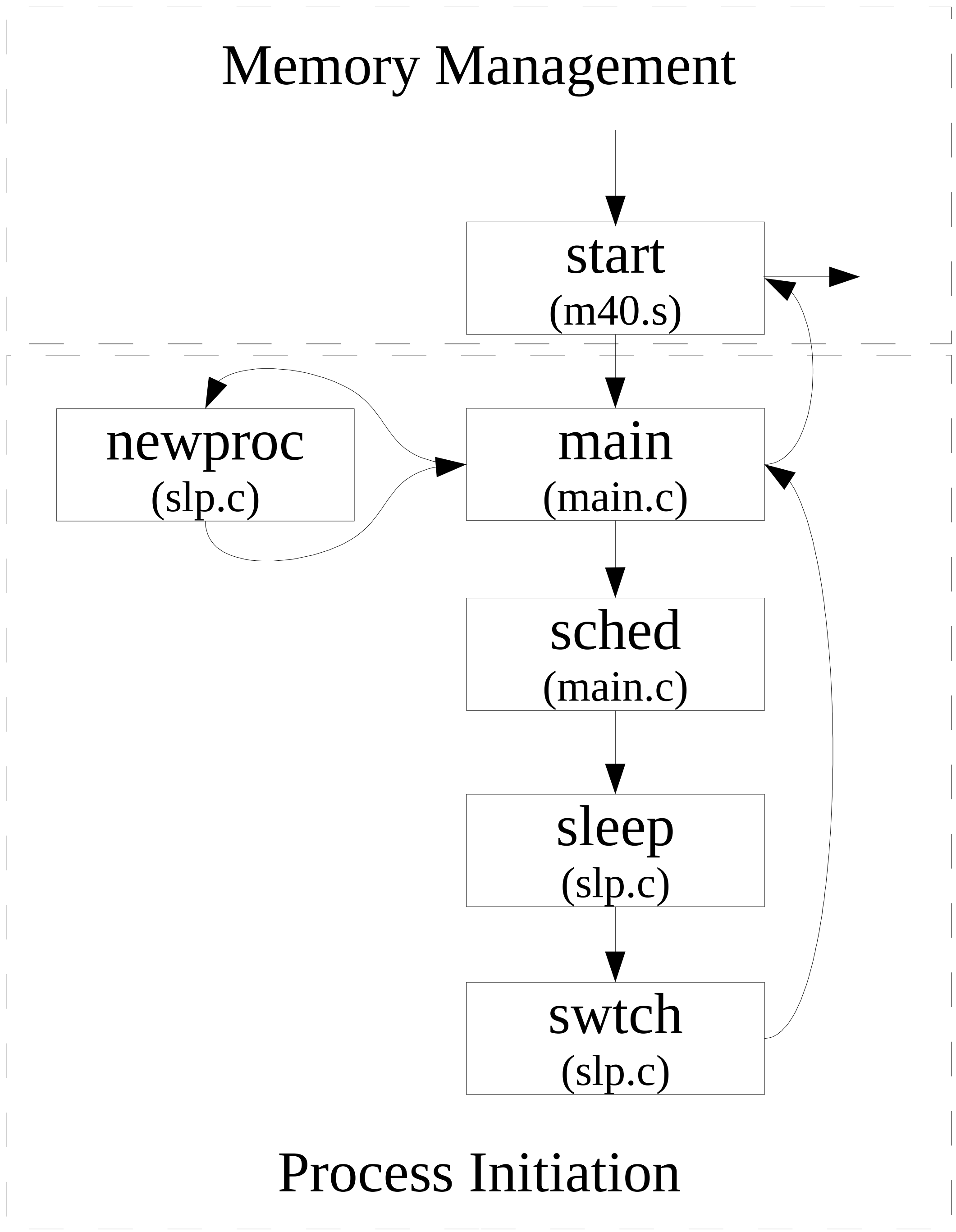
Outline

- Overview
- Initializiation
  - Initialize Kernel Segments
  - Enable Memory Management
- Process Initiation
  - Create System Process
  - Create Init Process
- Summary
- References

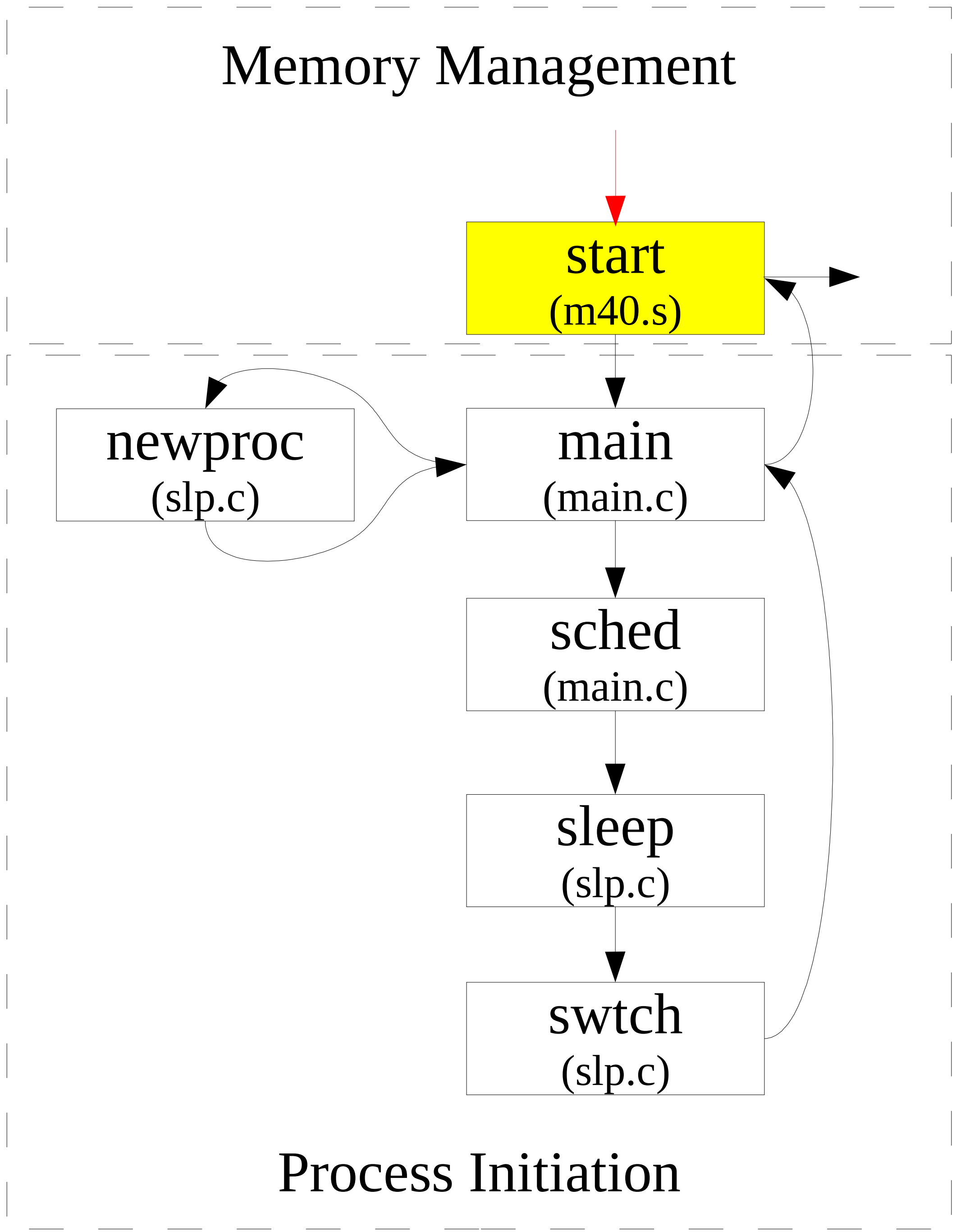
Overview



start
(m40.s)
newproc
(slp.c)
main
(main.c)
sched
(main.c)
sleep
(slp.c)
swtch
(slp.c)



Memory Management
start
(m40.s)
newproc
(slp.c)
main
(main.c)
sched
(main.c)
sleep
(slp.c)
swtch
(slp.c)
Process Initiation



Memory Management
start
(m40.s)
newproc
(slp.c)
main
(main.c)
sched
(main.c)
sleep
(slp.c)
swtch
(slp.c)
Process Initiation

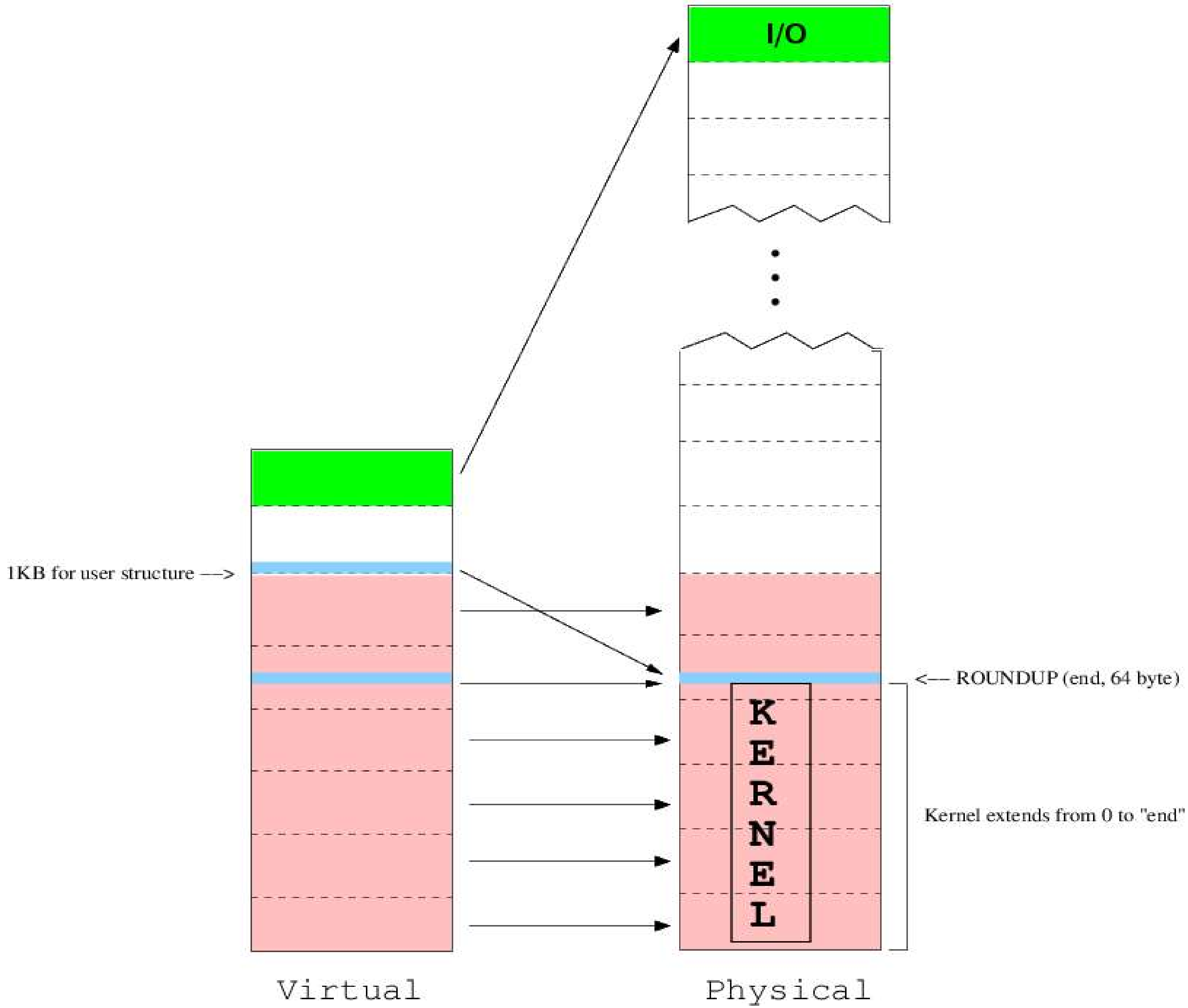
Memory Management
(start)

- Initialize kernel segments
- Initialize user segments
- Initialize I/O segments
- Start Memory Management Unit (MMU)

PDP-11/40 Memory

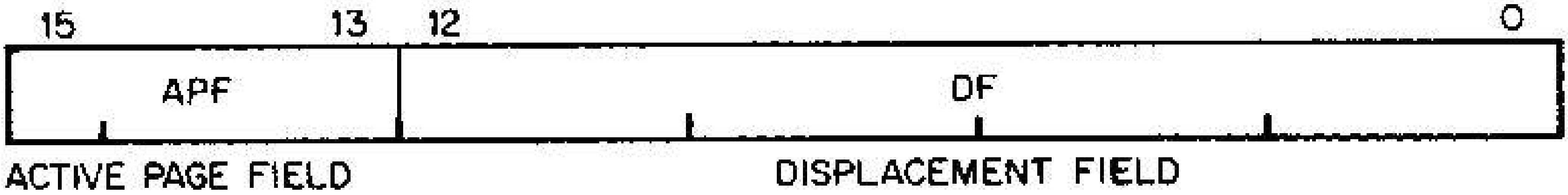
- 8 kernel mode memory pages
- 8 user mode memory pages
- 32 to 4096 word page length
  - (Unix uses 4096 word page length)
- 3 modes of memory access control

Kernel Address Space Mapping

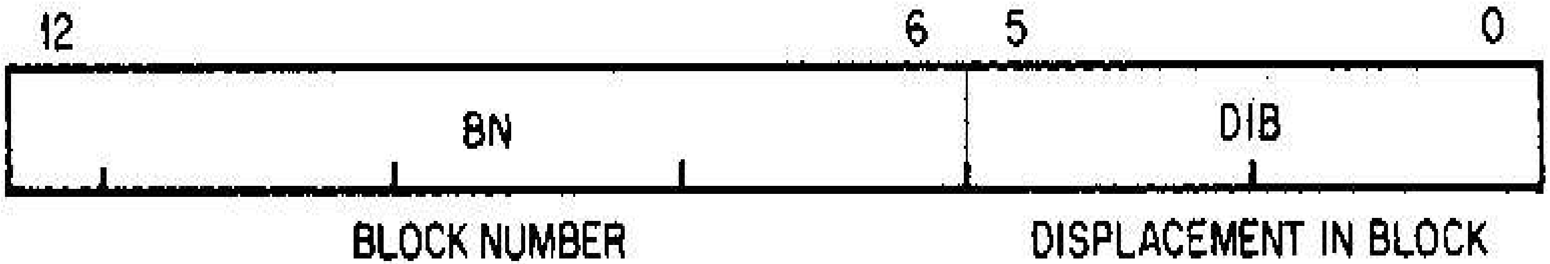


Virtual to Physical Address

(16 bits to 18 bits)

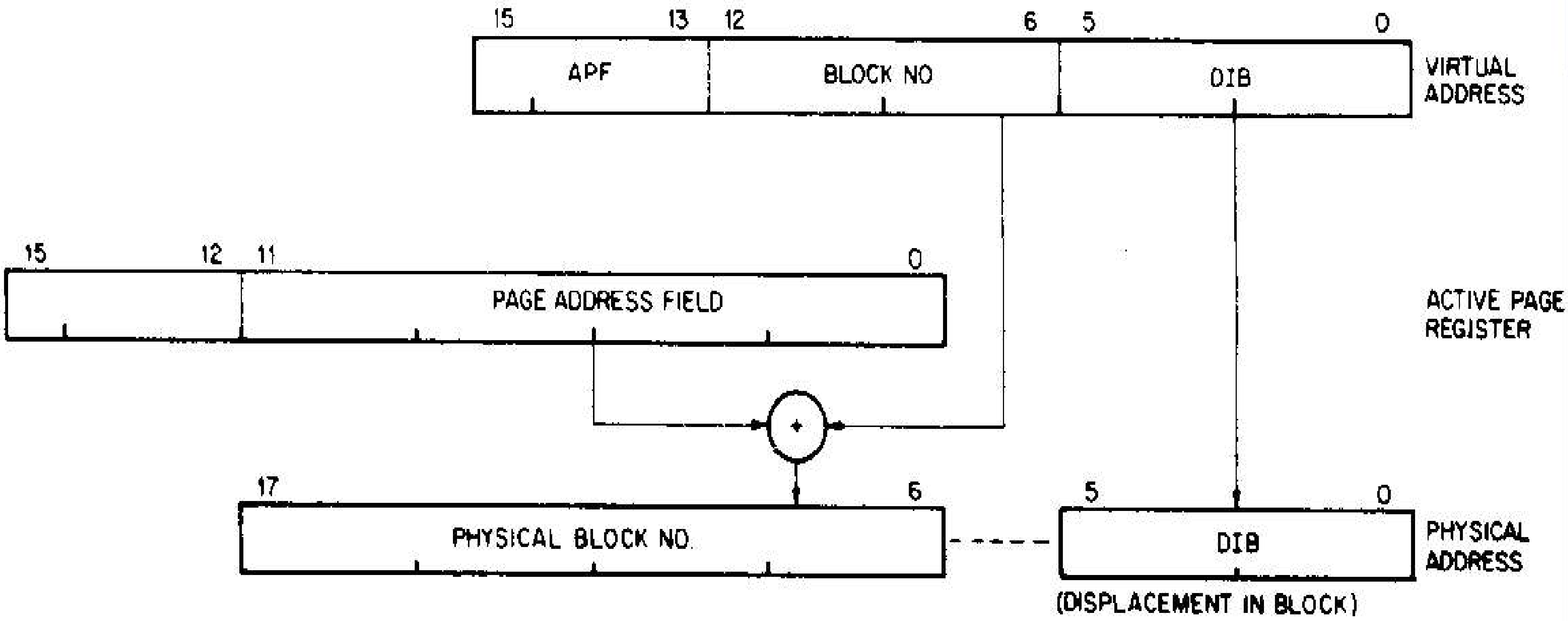


Virtual Address

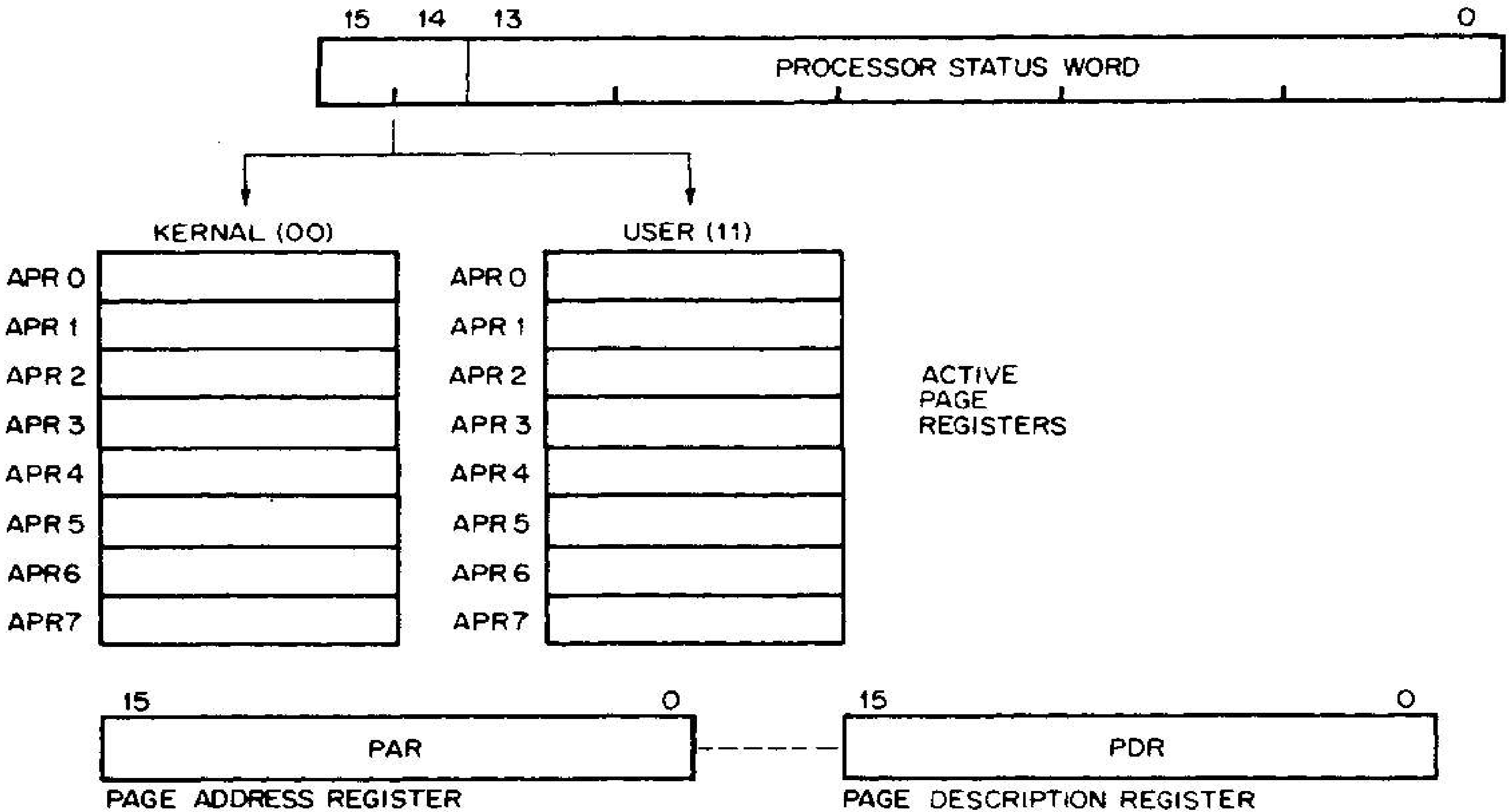


Displacement Field

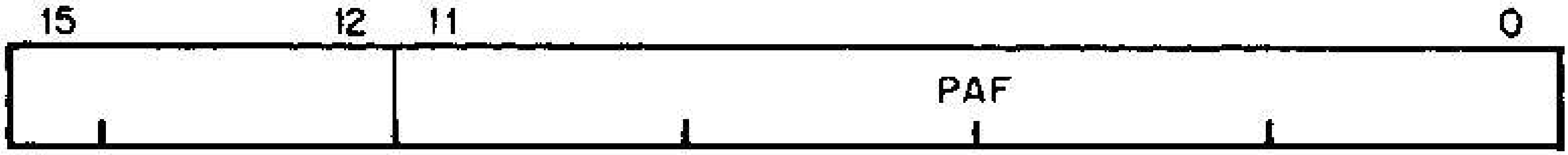
Construction of Physical Address



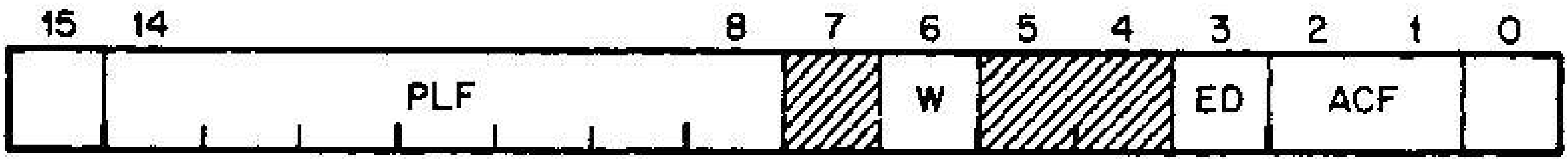
Active Page Registers (APR)



APR: PAR and PDR



PAR—Base Address



PDR—Page Description

Example

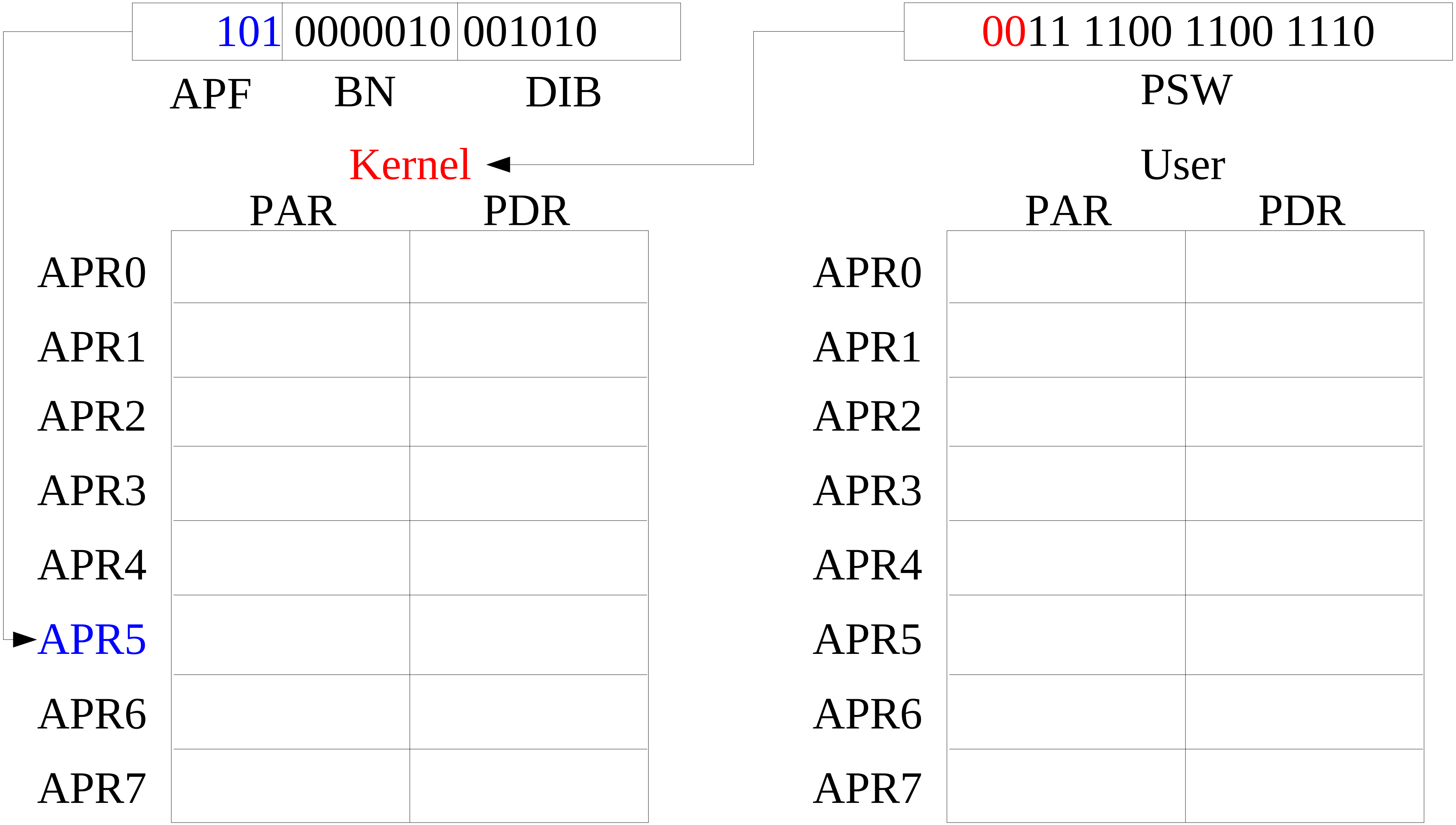
1010 0000 1000 1010

Virtual Address

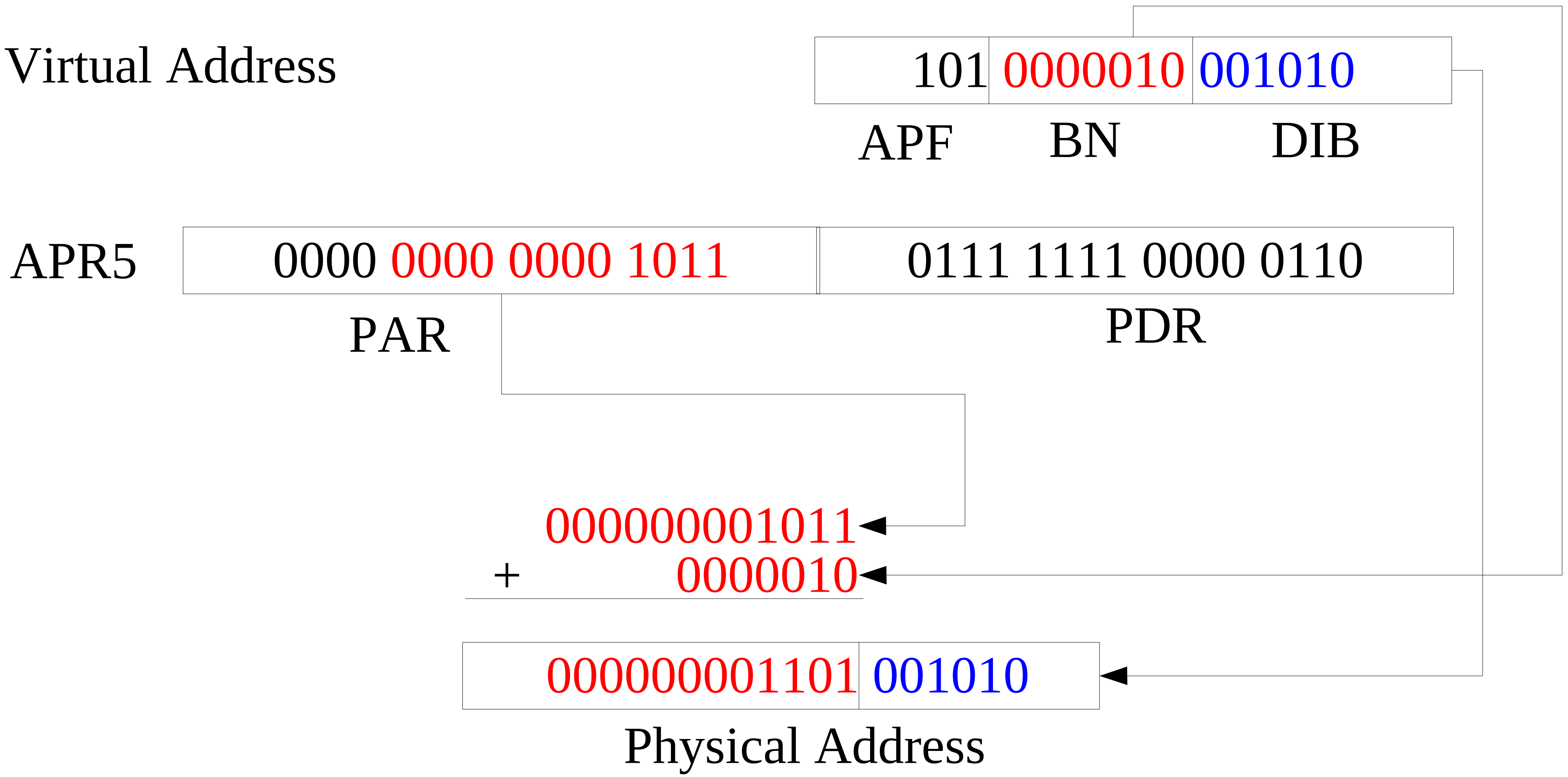
101	0000010001010
APF	DF

101	0000010	001010
APF	BN	DIB

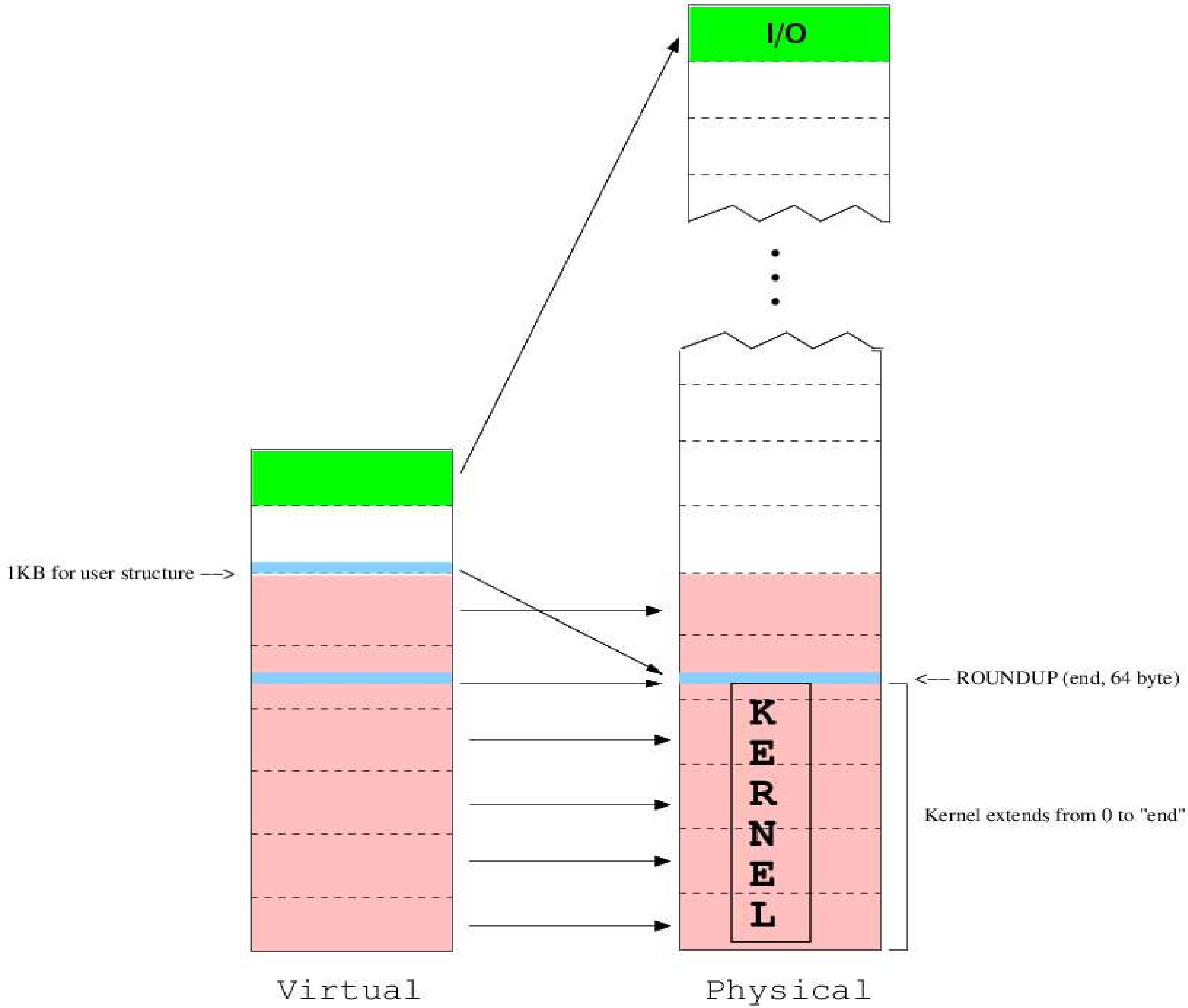
Example


101 0000010 001010
APF
BN
DIB
0011 1100 1100 1110
PSW
Kernel
User
PAR
PDR
APR0
APR1
APR2
APR3
APR4
APR5
APR6
APR7
PAR
PDR
APR0
APR1
APR2
APR3
APR4
APR5
APR6
APR7

Example


Virtual Address
101 0000010 001010
APF
BN
DIB
APR5
0000 0000 0000 1011
0111 1111 0000 0110
PAR
PDR
000000001011
+ 0000010
000000001101 001010
Physical Address

Kernel Address Space Mapping



Important Memory Locations

(m40.s)

- USIZE = 16 Size of User Block (*64 = 1024 B)
- PS = 177776 Program Status Word
- SSR0 = 177572 Status Register
- KISA0 = 172340 Kernel Segment Address Register #0
- KISA6 = 172354 Kernel Segment Address Register #6
- KISD0 = 172300 Kernel Segment Descriptor Register #0
- UISA0 = 177640 User Segment Address Register #0
- UISA1 = 177642 User Segment Address Register #1
- UISD0 = 177600 User Segment Descriptor Register #0
- UISD1 = 177602 User Segment Descriptor Register #1
- IO = 7600 I/O Segment Register

Initialize Kernel Segments

(start)

```
    mov   $KISA0, r0        start at first address and descriptor
    mov   $KISD0, r1        registers
    mov   $200, r4          increment pointer positions by 200 to
    clr   r2                0, 200, ..., 1200_8 blocks
    mov   $6, r3            for r6 = 6 to 0 (each segment register)
1:
    mov   r2, (r0)+         address set to current pointer position
    mov   $77406, (r1)+     descriptor set to 4K size and read-write
    add   r4, r2
    sob   r3, 1b
```

Initialize User Segment
(start)

```
mov   $_end + 63., r2
ash   $-6, r2
bic   $!1777, r2

mov   r2, (r0)+
mov   $USIZE - 1 \< 8 | 6, (r1)+
```

address (KISA6) set to mark end of program code and data area in user

descriptor set to 1024B size and read-write

_end rounded to multiple of 64

Right-shift bits by 6

Clear each bit in r2 that is set in !1777 (upper 6 bits)

Initialize I/O Segment

(start)

```
mov   $IO, (r0)+        address set to I/O segment
mov   $77406, (r1)+     descriptor set to 4K size and read-write
```

8th segment (KISA7) mapped into highest 4K word segment of the physical address space

Start MMU
(start)

mov	$_u + [USIZE*64.], sp	set stack pointer to highest word of per process data area
inc	SSR0	enable memory management

Memory management enabled when right-most bit of Status Register (SSR0) is set.

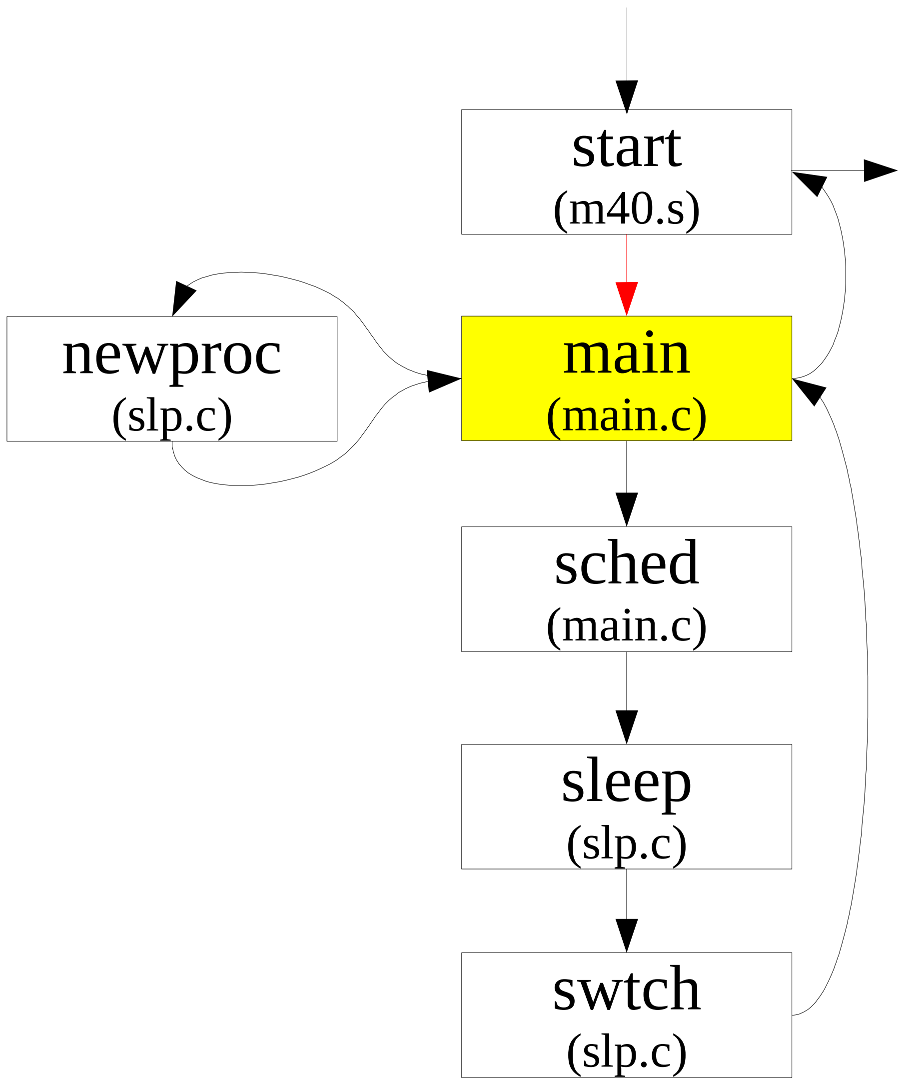
Call main
(start)

mov	$30000, PS	set previous mode of program status word to user ($30000_8 = 11\ 000...0_2$)
jsr	pc, _main	call main

```
mov   $170000, -(sp)
clr   -(sp)
rtt
```

main returns and does this much later...

Overview



start
(m40.s)
newproc
(slp.c)
main
(main.c)
sched
(main.c)
sleep
(slp.c)
swtch
(slp.c)

Initial Conditions in main

- Processor running at priority zero in kernel mode and with previous mode set to user
- Kernel mode segmentation registers set and MMU enabled
- All data areas used by the OS initialized
- Stack pointer (sp or r6) points to return address in start

Summary of main

1. Zero out and free all of core memory
2. Determine amount of memory available
3. Initialize swap space
4. Set up system/kernel process
5. Determine type of clock
6. Initialize buffer pools
7. Call newproc() to create second process
8. Call sched()

Summary of main

1. Zero out and free all of core memory
2. Determine amount of memory available
3. Initialize swap space
4. Set up system/kernel process
5. Determine type of clock
6. Initialize buffer pools
7. Call newproc() to create second process
8. Call sched()

How to handle processes?

In main, we need to:

- Set up system/kernel process
- Call newproc() to create second process
- Call sched()

First, need to define two structures

- proc
- user

Process Structure

(proc.h)

```
struct proc {
    p_stat;
    p_flag;
    p_pri;
    p_pid;
    p_ppid;
    p_addr;
    p_size;
    p_textp;
    ...
} proc[ NPROC ];
```

- Current state
- Process traits
- Priority
- User ID
- Parent ID
- Address of program's data segment
- Size of program image in blocks
- Pointer to text segment

User Structure

(user.h)

```
struct user {
    u_rsav;
    u_qsav;
    u_ssav;
    u_procp;
    u_uisa[16];
    u_uisd[16];
    u_tsize;
    u_dsize;
    u_ssize;
    ...
} u;
```

- Arrays for storing registers r5 and r6 (environment and sp)
- Address of corresponding proc structure
- User page address and description registers
- Size of text, data, and stack segments

User Structure
(user.h)

35 fields in total also concerned with:

- Saving floating point registers
- User identification
- Parameters for input/output operations
- File access control
- System call parameters
- Accounting information

Proc Versus User

proc

- One allocated per process
- Never swapped from memory
- Must be available any time
- Points to user

```
struct proc {
    ...
} proc[ NPROC ];
```

user

- One allocated per process
- May be swapped when not running
- Only one available at a time
- Points to proc

```
struct user {
    ...
} u;
```

Main Continued

In main, we can now:

- Set up system/kernel process
- Call newproc() to create second process
- Call sched()

Set up system process
(main.c)

```
proc[0].p_addr = *ka6;

proc[0].p_size  = USIZE;

proc[0].p_stat  = SRUN;
proc[0].p_flag |= SLOAD;
proc[0].p_flag |= SSYS;

u.u_procp = &proc[0];
```

- Set address of process's location in memory
- Set size of process's segment
- Mark the process as runnable, in memory, and should not be swapped out

- Save its proc position

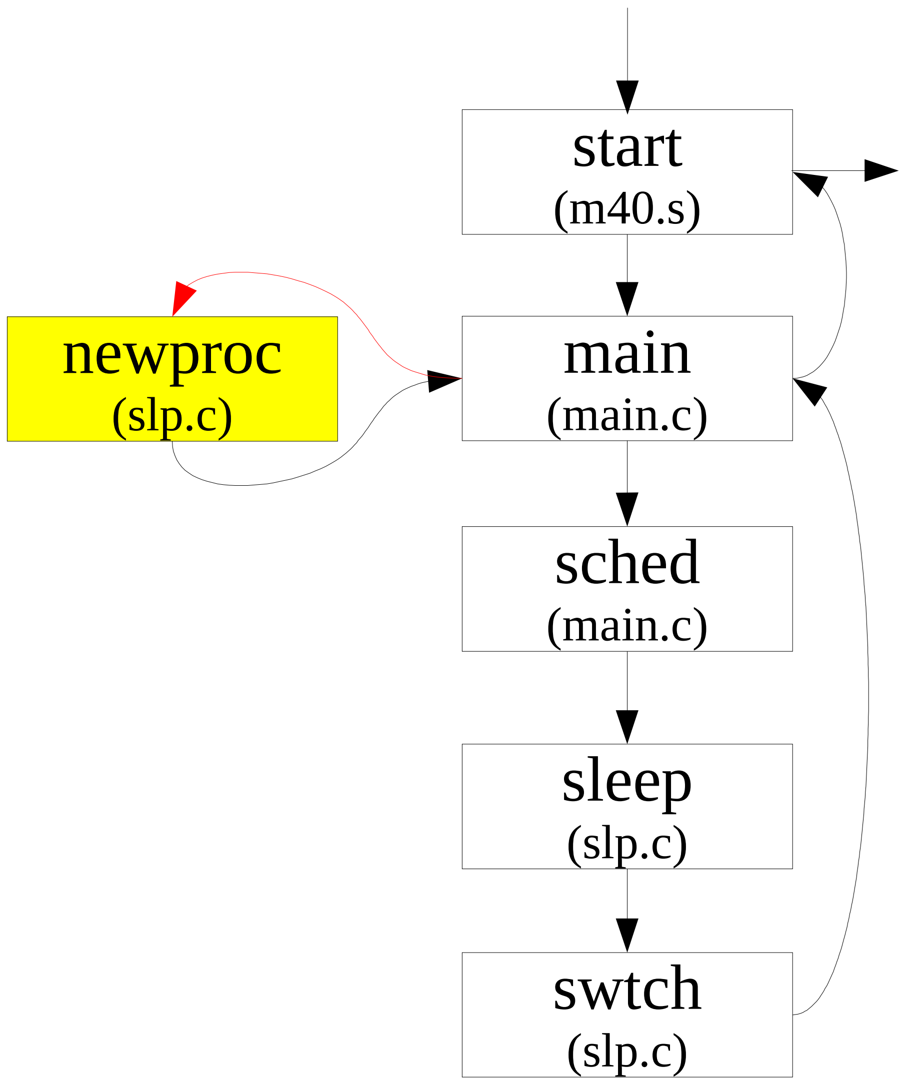
Main Again

In main, we can now:

- Set up system/kernel process
- Call newproc() to create second process
- Call sched()

```
if( newproc() )
{
    expand( USIZE + 1 );
    estabur( 0, 1, 0, 0 );
    copyout( icode, 0, sizeof( icode ) );
    return;
}
sched();
```

Overview



start
(m40.s)
newproc
(slp.c)
main
(main.c)
sched
(main.c)
sleep
(slp.c)
swtch
(slp.c)

newproc

- Initialize second “proc” structure—proc[1]
  - Locate unused slot in proc table
  - Copy proc[0]'s fields into proc[1]
  - Save environment and stack pointers into u.u_rsav
  - Allocate data area in memory for proc[1]
  - Copy proc[1]'s data area (including u.u_rsav) into proc[1]'s data area
  - Set proc[1]'s “u.u_procp” to &proc[1]

- Exact copy of proc[0] made, except value of “u.u_procp” in proc[1] is “&proc[1]”
- Return 0

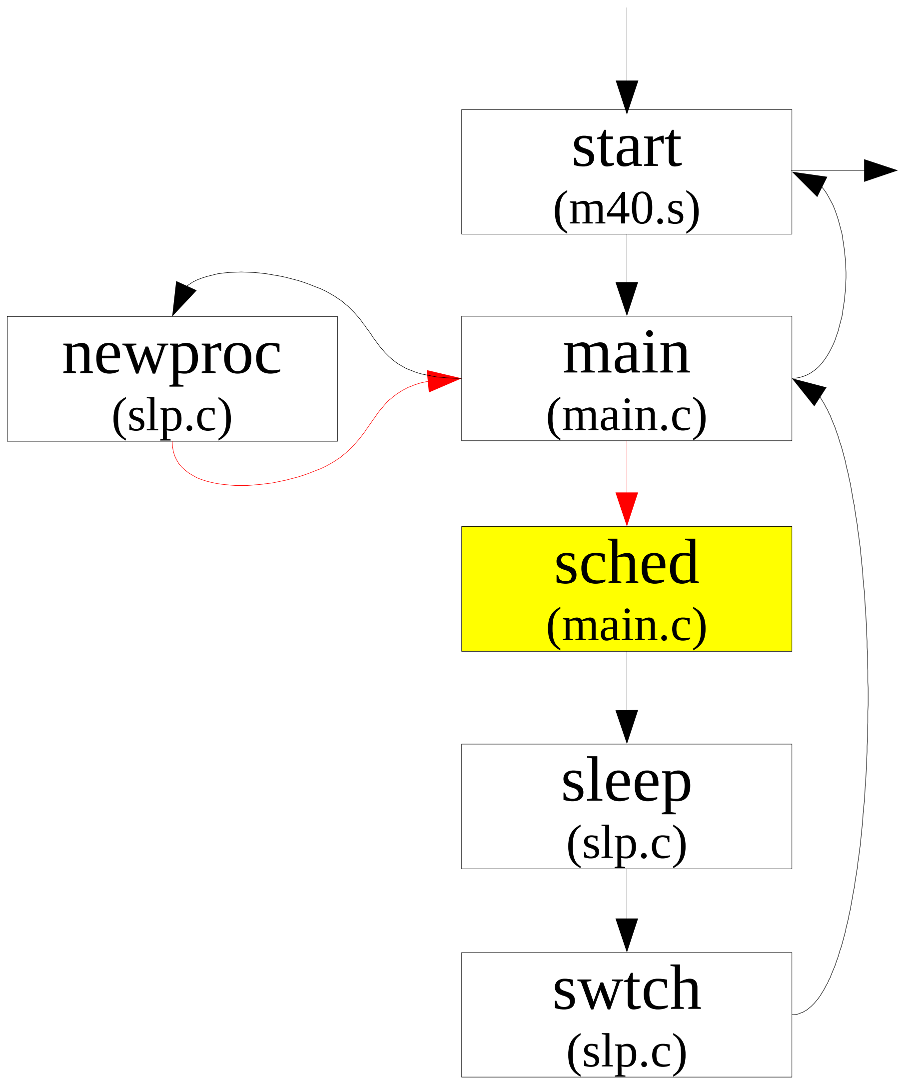
Main Again

In main, we can now:

- Set up system/kernel process
- Call newproc() to create second process
- Call sched()

```
if( newproc() )          <-- 0 returned
{
    expand( USIZE + 1 );
    estabur( 0, 1, 0, 0 );
    copyout( icode, 0, sizeof( icode ) );
    return;
}
sched();
```

Overview


start
(m40.s)
newproc
(slp.c)
main
(main.c)
sched
(main.c)
sleep
(slp.c)
swtch
(slp.c)

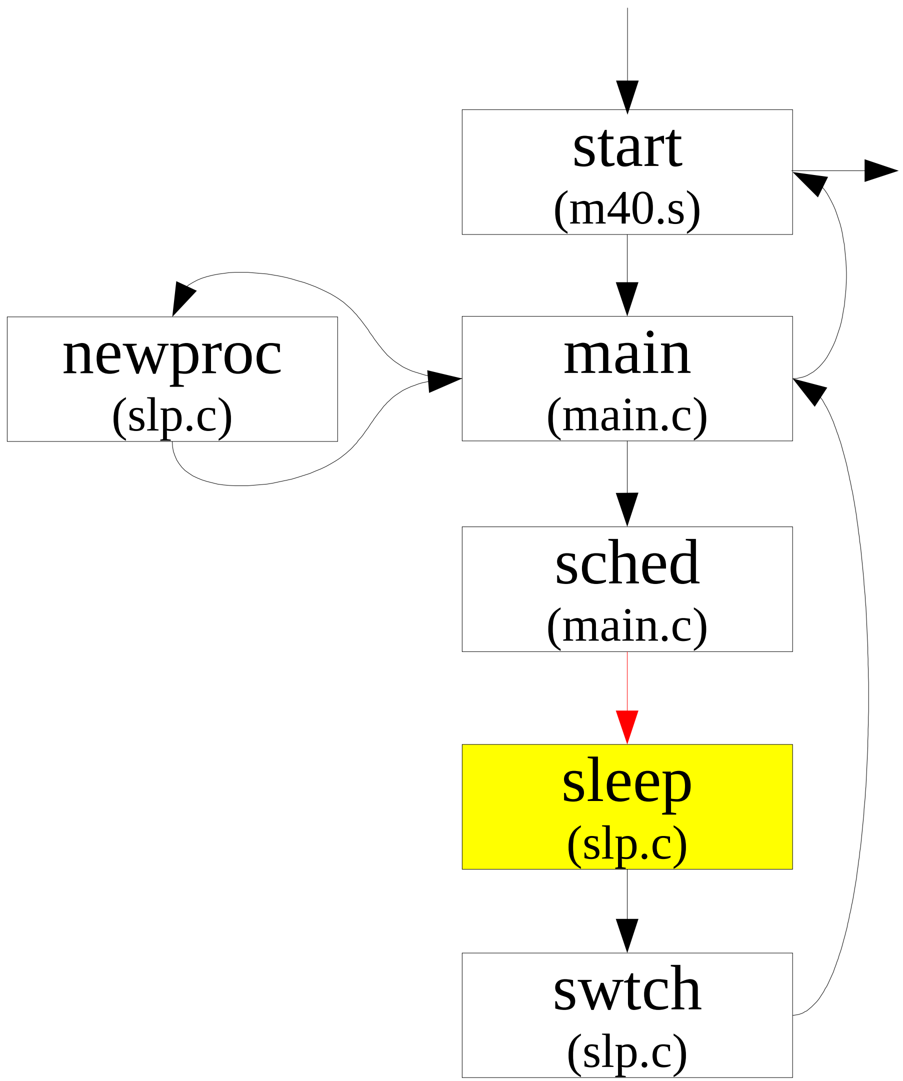
Scheduling the First Process

(sched)

See if anyone wants to be swapped in
swap out processes until there is room

- Disable interrupts
- Find process that is runnable but not in memory
  - Search fails the first time—proc[0] and proc[1] are the only processes and are both in memory
  - Call sleep() to wait for runnable, swapped process (priority set to max = -100)

Overview

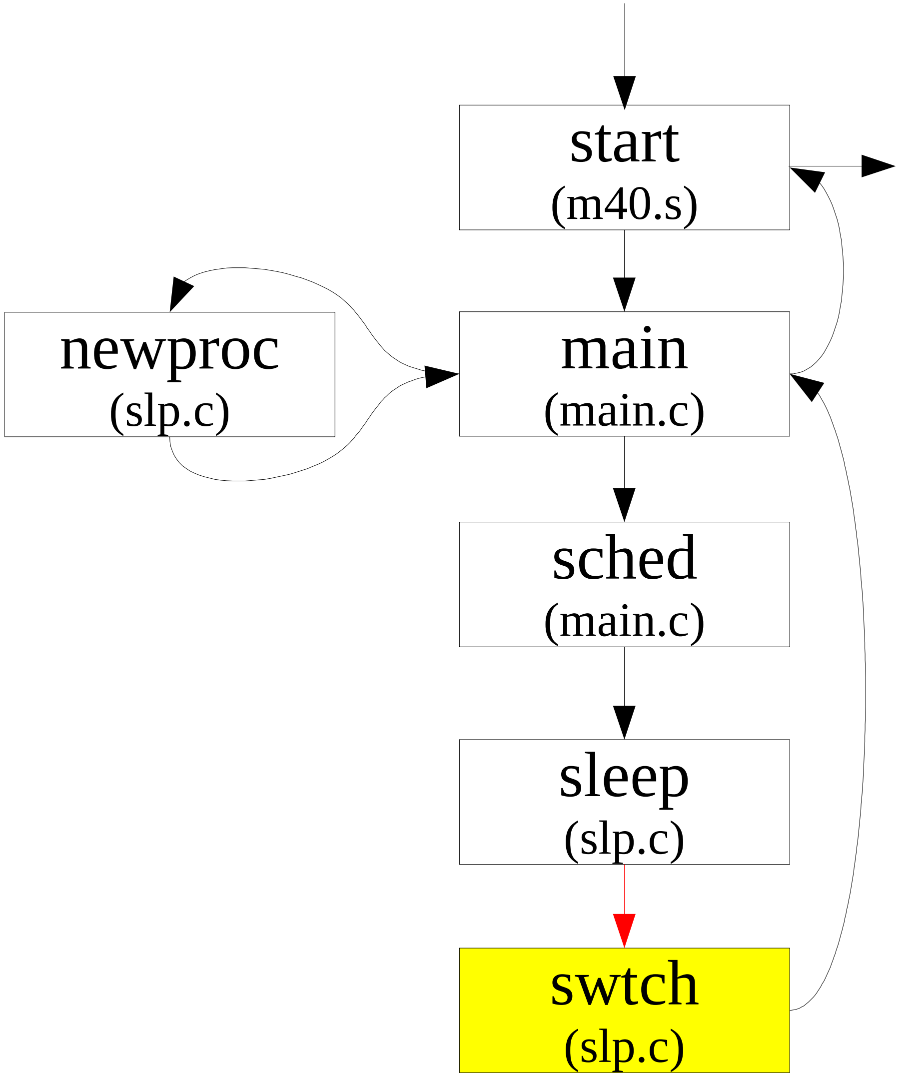


start
(m40.s)
newproc
(slp.c)
main
(main.c)
sched
(main.c)
sleep
(slp.c)
swtch
(slp.c)

sleep

- Save PSW
- Get current process—proc[0]
- Priority is negative, so set status of current process to sleep
- Call swtch()

Overview


start
(m40.s)
newproc
(slp.c)
main
(main.c)
sched
(main.c)
sleep
(slp.c)
swtch
(slp.c)

swtch

- Static variable p set to process 0 initially (value preserved between calls)
- Call savu() to save the stack pointer and the environment pointer for the current process in u.u_rsav
- Call retu() on proc[0] (context switch from 0 to 0 the first time)
  - Reset kernel address register for segment #6 to value passed as an argument
  - Reset the stack and environment pointers to values appropriate to the revised current process, whose execution is about to be resumed

swtch (2)

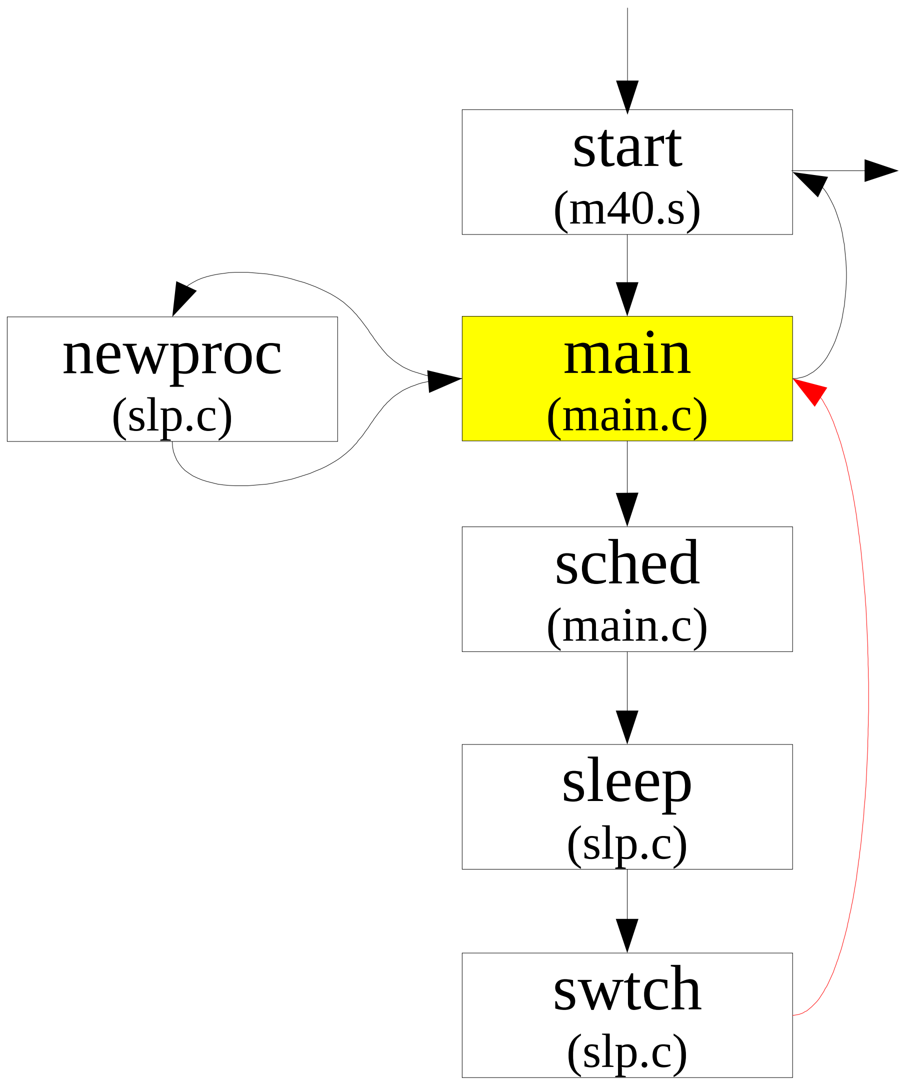
- Search for highest priority, runnable process (starting at end)
- Process 1 found
- retu() called to switch to process 1
- Call sureg() to copy appropriate values for current process into user mode segmentation registers, which were stored earlier in u.u_uisa and u.u_uisd (0s copied the first time)
- Return 1... to main! (How?!)

newproc

- Initialize second “proc” structure—proc[1]
  - Locate unused slot in proc table
  - Copy proc[0]'s fields into proc[1]
  - Save environment and stack pointers into u.u_rsav
  - Allocate data area in memory for proc[1]
  - Copy proc[1]'s data area (including u.u_rsav) into proc[1]'s data area
  - Set proc[1]'s “u.u_procp” to &proc[1]

- Exact copy of proc[0] made, except value of “u.u_procp” in proc[1] is “&proc[1]”
- Return 0

Overview



start
(m40.s)
newproc
(slp.c)
main
(main.c)
sched
(main.c)
sleep
(slp.c)
swtch
(slp.c)

Main Again

In main, we can now:

- Set up system/kernel process
- Call newproc() to create second process
- Call sched()

```
if( newproc() )                    ◄──────────── 1 returned
{
    expand( USIZE + 1 );
    estabur( 0, 1, 0, 0 );
    copyout( icode, 0, sizeof( icode ) );
    return;
}
sched();
```

main Again

- Call expand() to allocate new, larger area for process 1 and copy original data into it (no original data for proc[1])
- Call estabur() to set prototype segmentation registers (1 data segment for proc[1])
- Call copyout() to copy the array “icode” into start of user space
- Return to start in m40.s after the jump to main()

main Again

- Call expand() to allocate new, larger area for process 1 and copy original data into it (no original data for proc[1])
- Call estabur() to set prototype segmentation registers (1 data segment for proc[1])
- Call copyout() to copy the array “icode” into start of user space
- Return to start in m40.s after the jump to main()

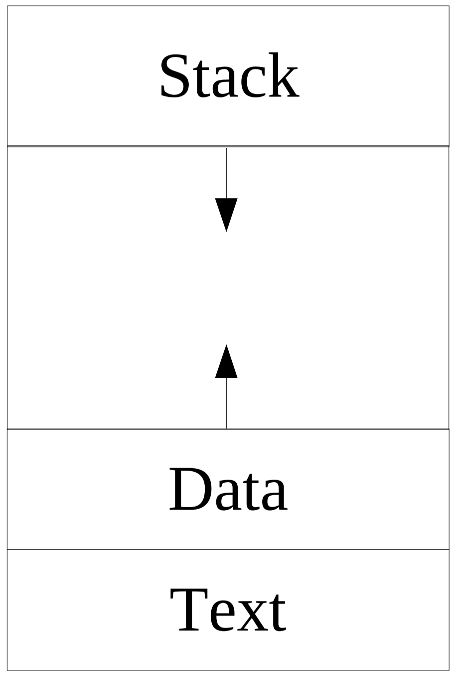
estabur

Set up prototype segmentation registers:

- u.u_uisa
- u.u_uisd

Segments created:

- Text—Read-only
- Data—Read-write
- Stack—Read-write, expands down

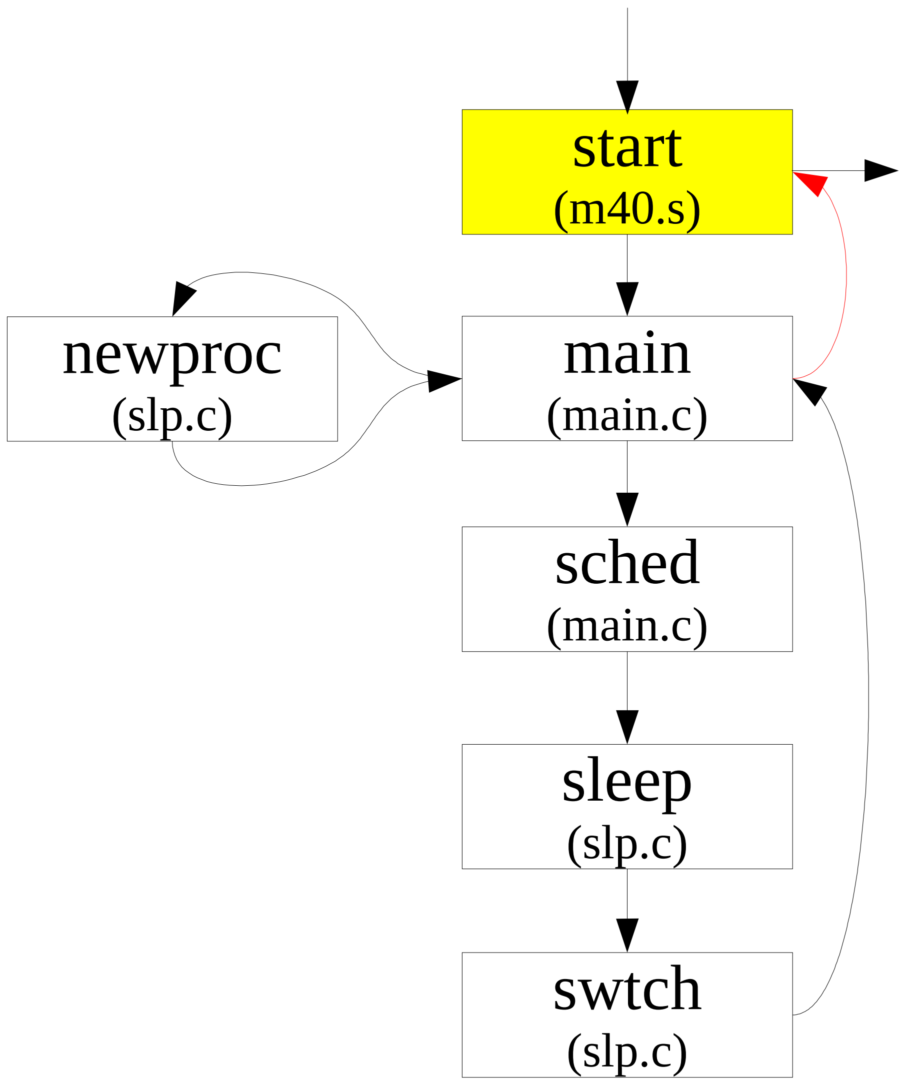


Virtual Address Space

main Again

- Call expand() to allocate new, larger area for process 1 and copy original data into it (no original data for proc[1])
- Call estabur() to set prototype segmentation registers (1 data segment for proc[1])
- Call copyout() to copy the array “icode” into start of user space
- Return to start in m40.s after the jump to main()

Overview


start
(m40.s)
newproc
(slp.c)
main
(main.c)
sched
(main.c)
sleep
(slp.c)
swtch
(slp.c)

start Again

- Execute in user mode the instruction at user mode address 0— icode (/etc/init)

```
mov   $170000, -(sp)
clr   -(sp)
rtt
```

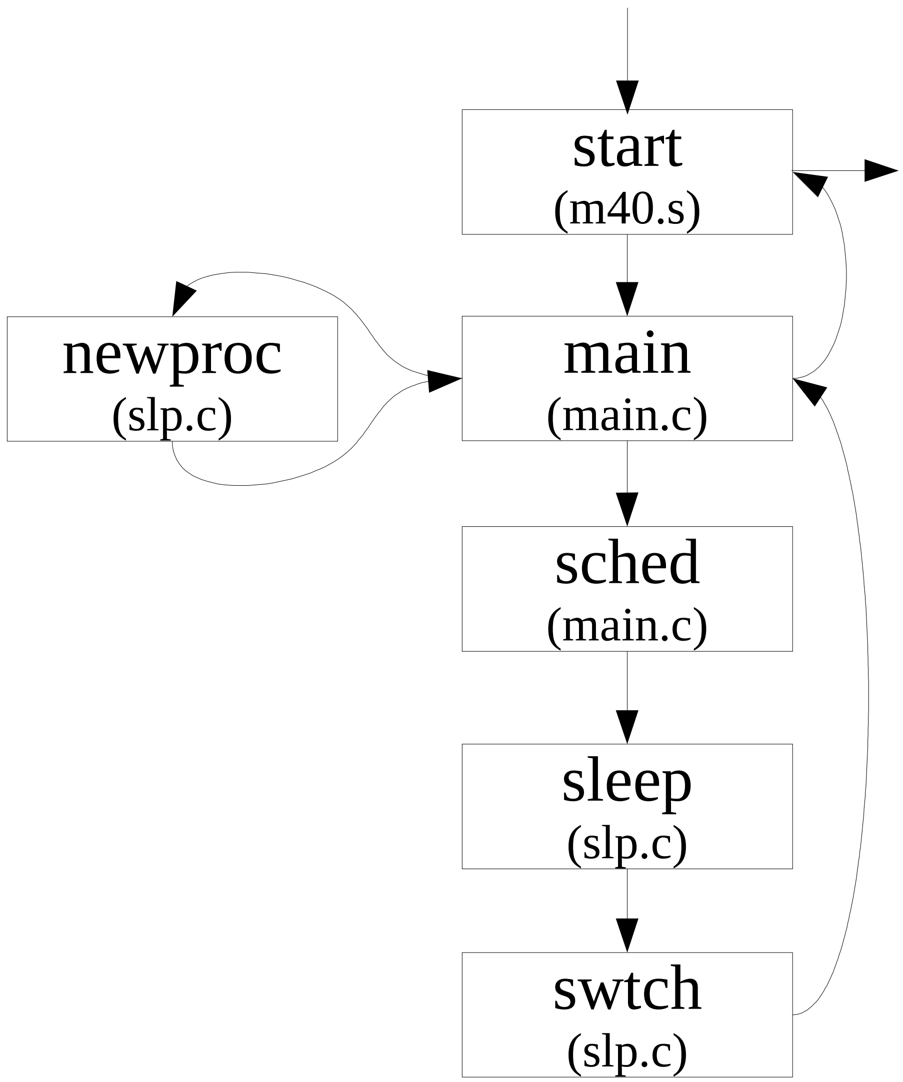
Execute icode—init program (main.c)

```
int icode[] =
{
    0104413,  // sys exec; init; initp
    0000014,
    0000010,
    0000777,  // br .
    0000014,  // initp: init; 0
    0000000,
    0062457,  // init: </etc/init\0>
    0061564,
    0064457,
    0064556,
    0000164
};
```

Equivalent C Program

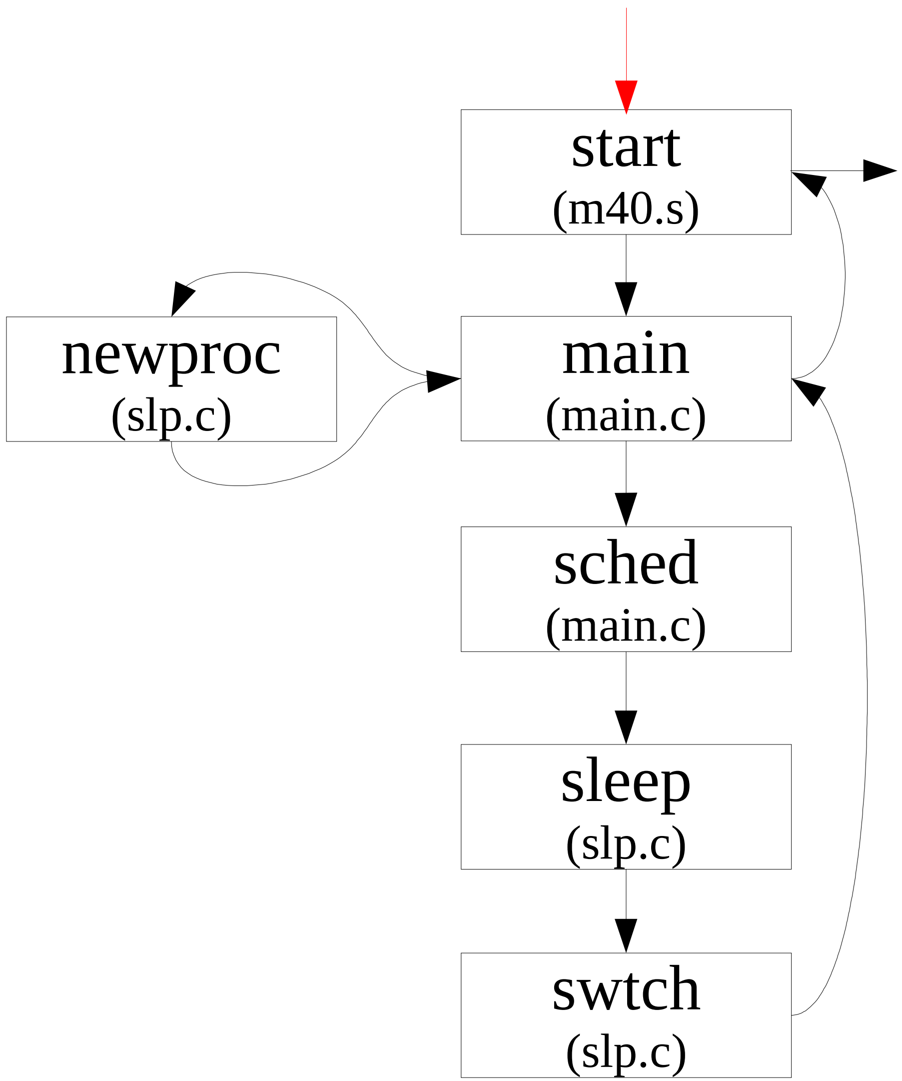
```
char* init = “/etc/init”;
main( ) {
    execl( init, init, 0 );
    while( 1 ) ;
}
```

Summary



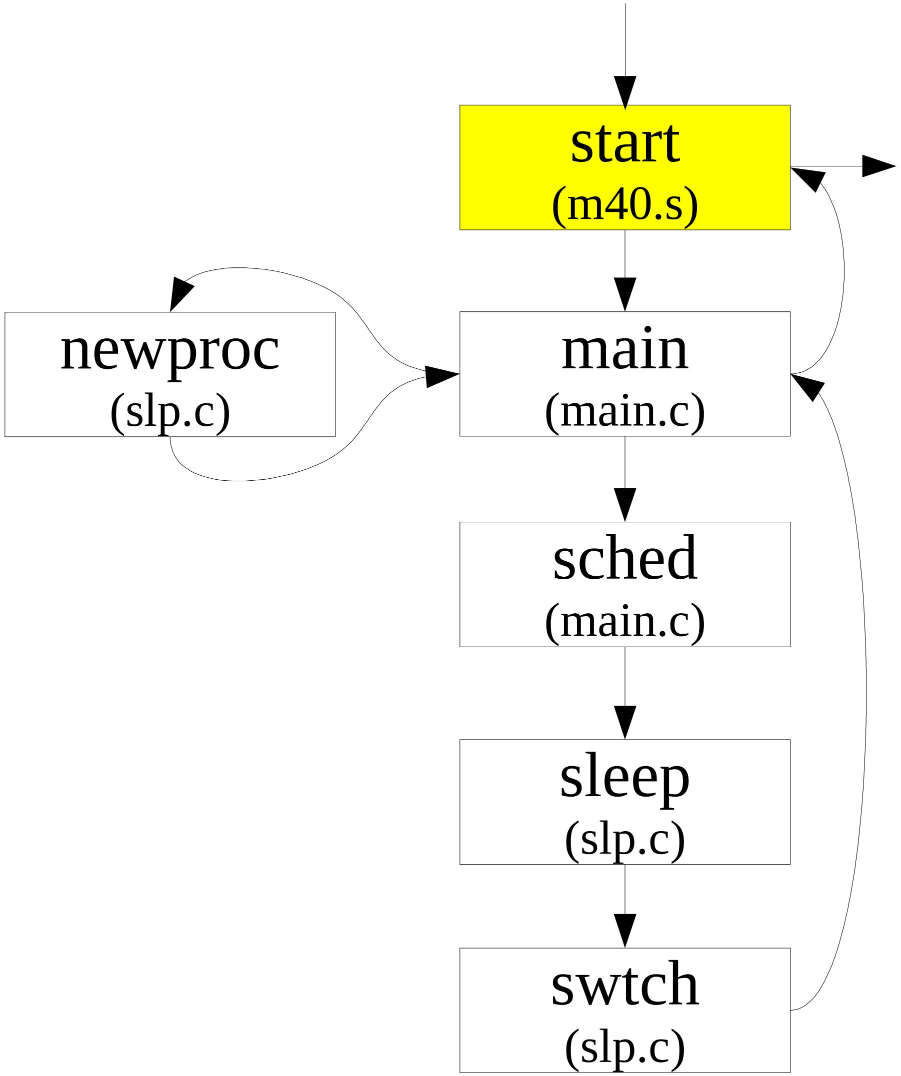
start
(m40.s)
newproc
(slp.c)
main
(main.c)
sched
(main.c)
sleep
(slp.c)
swtch
(slp.c)

Summary



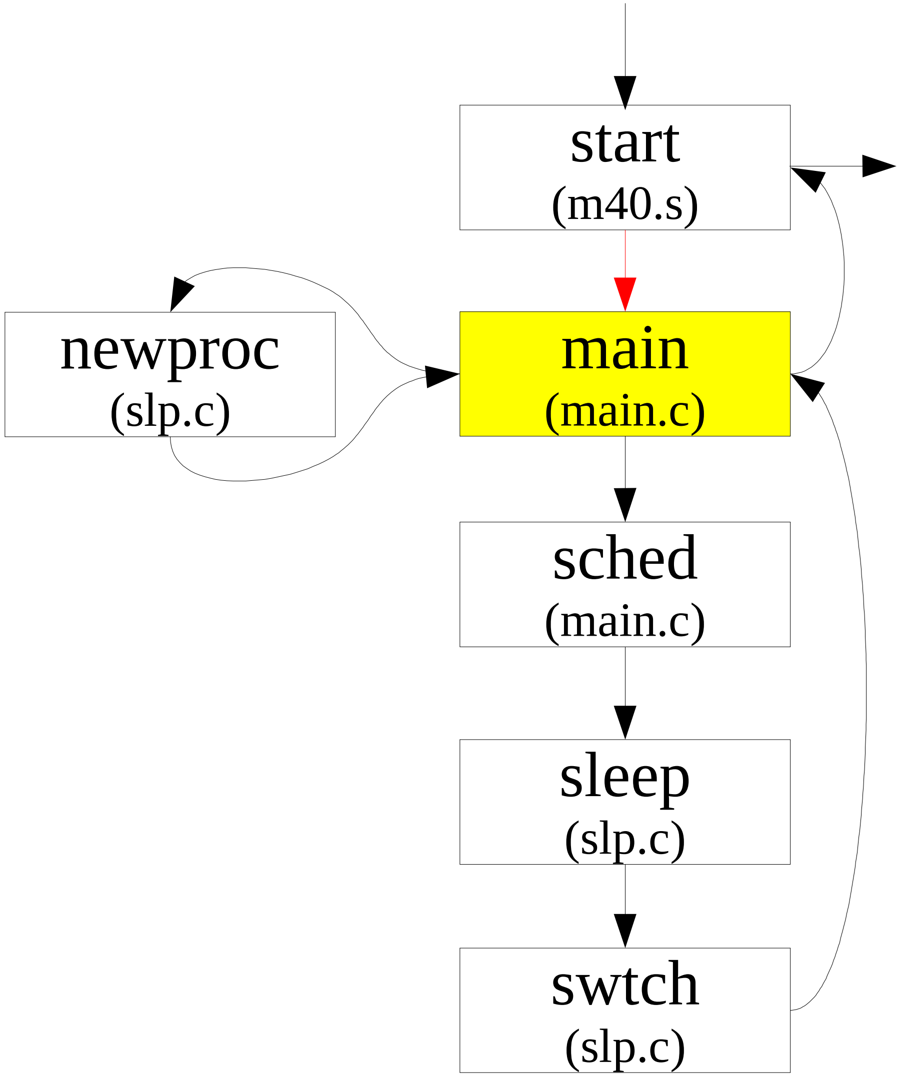
1. User turns computer on

Summary



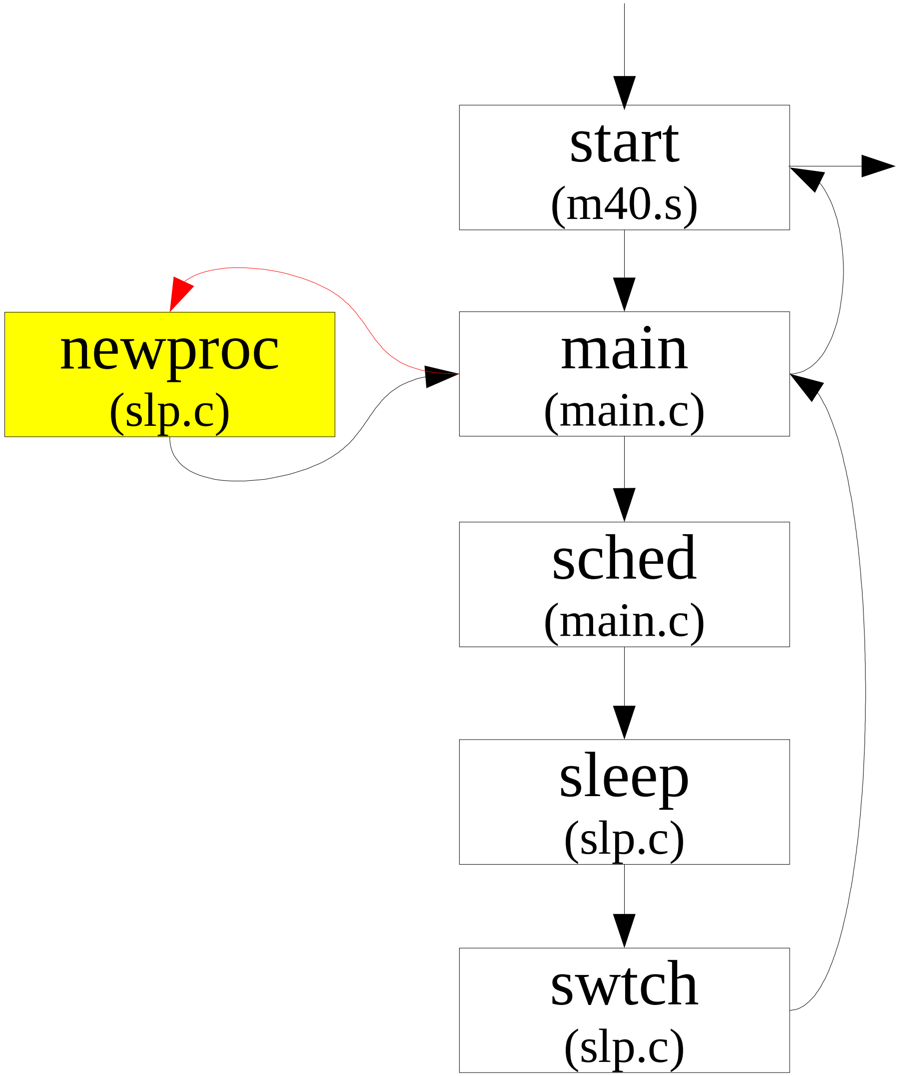
1. User turns computer on
2. Segments initialized
3. Memory management enabled

Summary



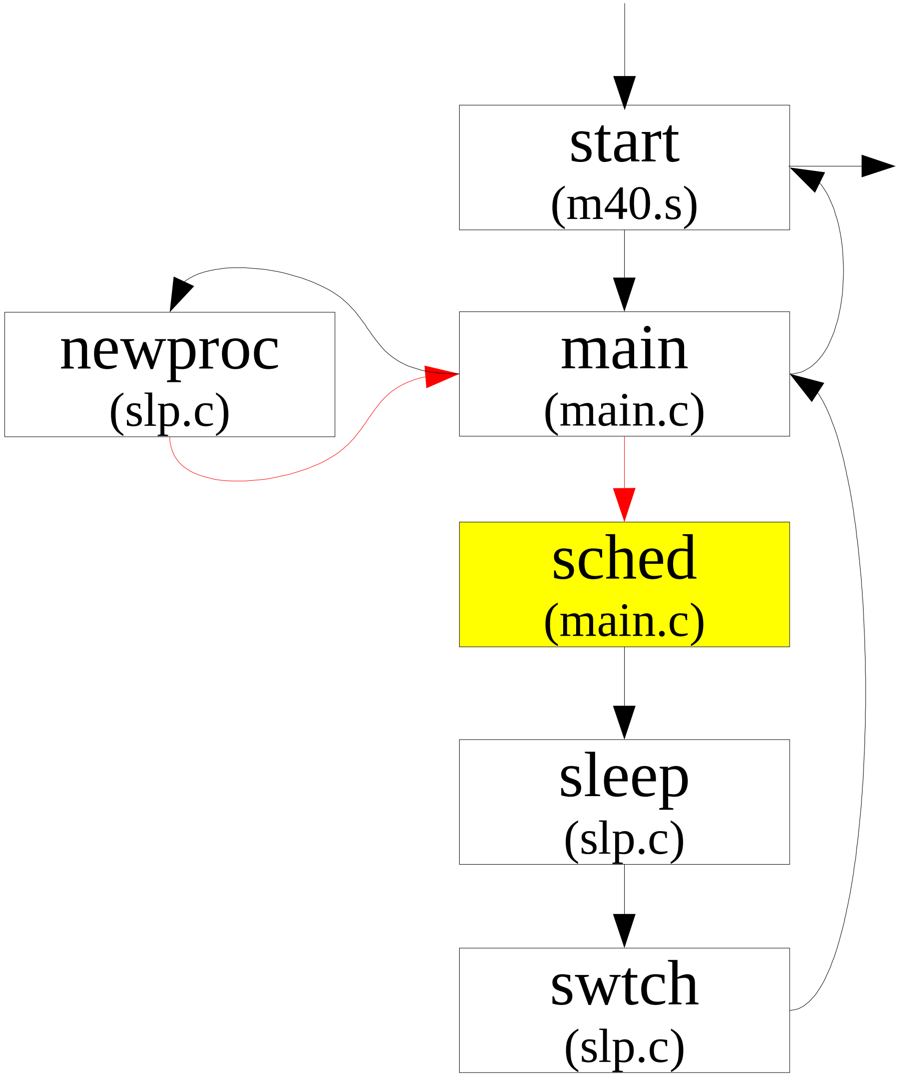
1. User turns computer on
2. Segments initialized
3. Memory management enabled
4. Initialize various components
5. Initialize system process (#0)

Summary



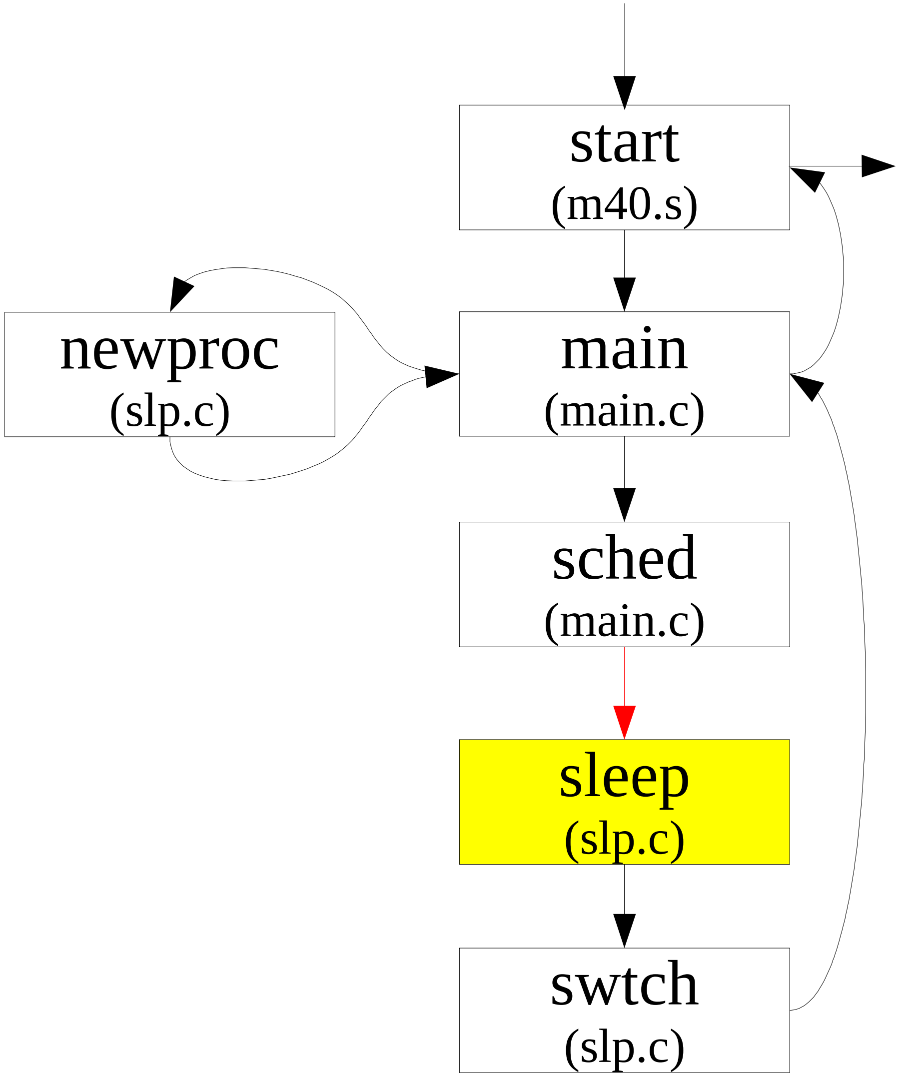
1. User turns computer on
2. Segments initialized
3. Memory management enabled
4. Initialize various components
5. Initialize system process (#0)
6. Copy system process (0 -> 1)

Summary



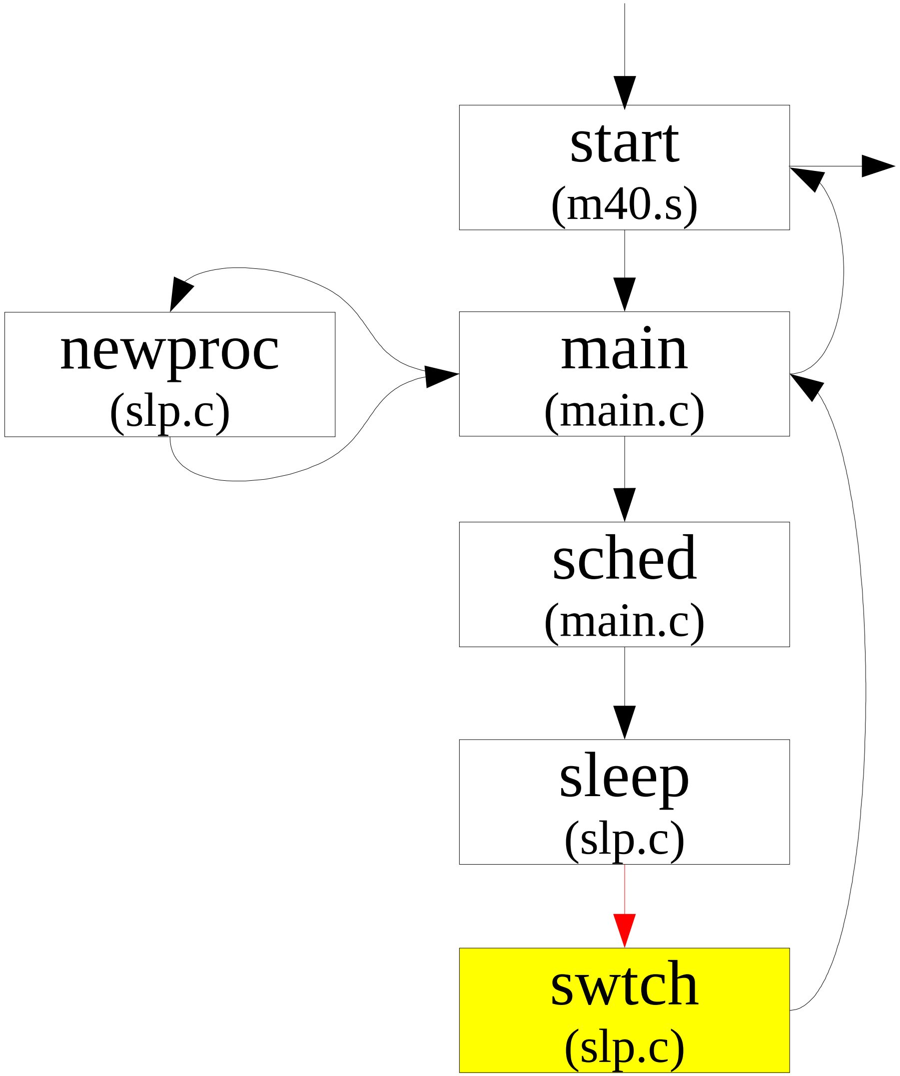
1. User turns computer on
2. Segments initialized
3. Memory management enabled
4. Initialize various components
5. Initialize system process (#0)
6. Copy system process (0 -> 1)
7. Scheduler/Disable Interrupts

Summary



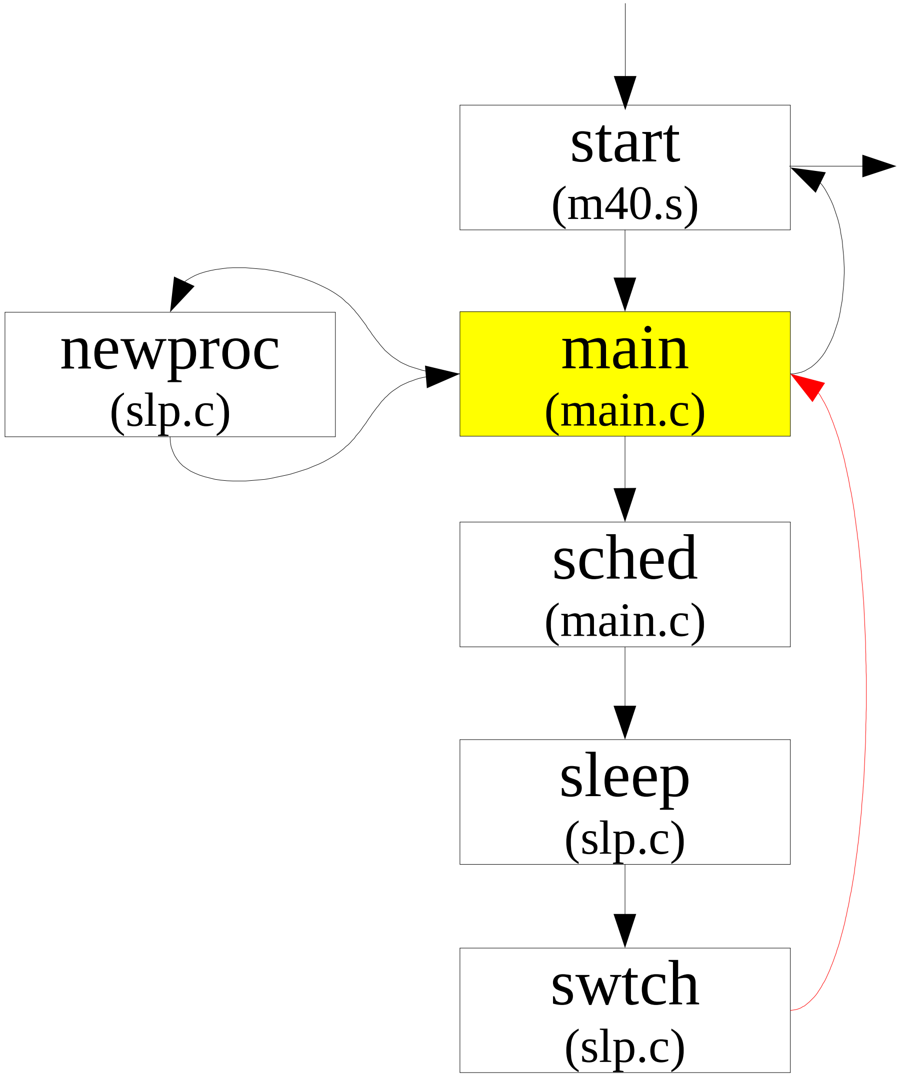
1. User turns computer on
2. Segments initialized
3. Memory management enabled
4. Initialize various components
5. Initialize system process (#0)
6. Copy system process (0 -> 1)
7. Scheduler/Disable Interrupts
8. proc[0] goes to sleep

Summary



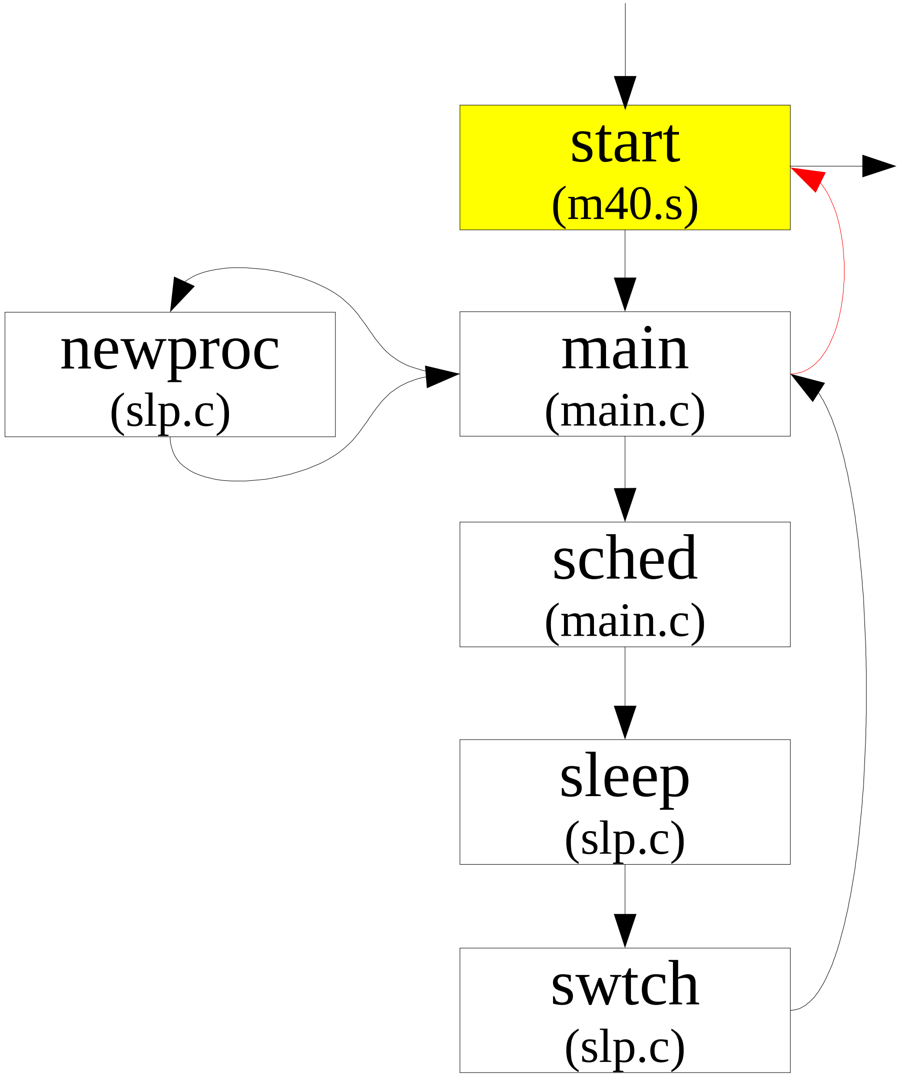
1. User turns computer on
2. Segments initialized
3. Memory management enabled
4. Initialize various components
5. Initialize system process (#0)
6. Copy system process (0 -> 1)
7. Scheduler/Disable Interrupts
8. proc[0] goes to sleep
9. Switch to proc[1] (old proc[0])

Summary



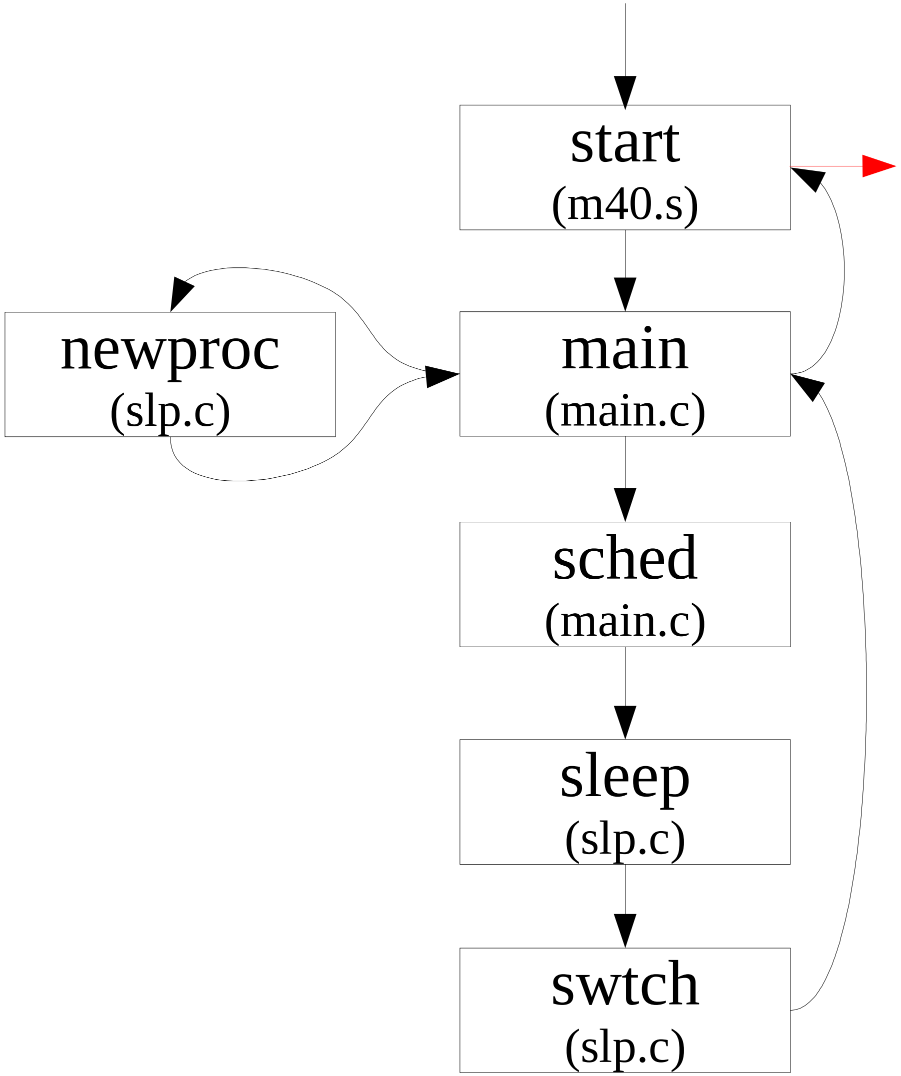
1. User turns computer on
2. Segments initialized
3. Memory management enabled
4. Initialize various components
5. Initialize system process (#0)
6. Copy system process (0 -> 1)
7. Scheduler/Disable Interrupts
8. proc[0] goes to sleep
9. Switch to proc[1] (old proc[0])
10. Load "init" into proc[1]

Summary



1. User turns computer on
2. Segments initialized
3. Memory management enabled
4. Initialize various components
5. Initialize system process (#0)
6. Copy system process (0 -> 1)
7. Scheduler/Disable Interrupts
8. proc[0] goes to sleep
9. Switch to proc[1] (old proc[0])
10. Load “init” into proc[1]
11. Run init

Summary



1. User turns computer on
2. Segments initialized
3. Memory management enabled
4. Initialize various components
5. Initialize system process (#0)
6. Copy system process (0 -> 1)
7. Scheduler/Disable Interrupts
8. proc[0] goes to sleep
9. Switch to proc[1] (proc[0] copy)
10. Load “init” into proc[1]
11. Run init
12. Initialization complete!

References

6.828 / Fall 2004. 6.828: Operating System Engineering. 12 March 2005 <http://www.pdos.lcs.mit.edu/6.828/2004/index.html>.

John Lions. *Lions' Commentary on UNIX 6th Edition with Source Code.* Peer-to-Peer Communications, 1996.

PDP11/40 Processor Handbook. Digital Equipment Corporation. 1972.