

Started on Tuesday, 16 January 2024, 5:08 PM

State Finished

Completed on Tuesday, 16 January 2024, 5:52 PM

Time taken 44 mins 52 secs

Grade 10.20 out of 15.00 (68%)

Question 1

Incorrect

Mark 0.00 out of 1.00

Select all the correct statements about the process init on Linuxes/Unixes.

Select one or more:

- a. init can not be killed with SIGKILL ✓
- b. only a process run by 'root' user can exec 'init'
- c. init is created by kernel 'by hand'
- d. any user process can fork and exec init ✗
- e. init typically has a pid=1 ✓
- f. init is created by kernel by forking itself ✗
- g. no process can exec 'init'

Your answer is incorrect.

The correct answers are: init is created by kernel 'by hand', init typically has a pid=1, init can not be killed with SIGKILL, only a process run by 'root' user can exec 'init'

Question 2

Incorrect

Mark 0.00 out of 1.00

Write the possible contents of the file /tmp/xyz after this program.

In the answer if you want to mention any non-text character, then write \0. For example abc\0\0 means abc followed by any two non-text characters

```
int main(int argc, char *argv[]) {  
    int fd1, fd2, n, i;  
    char buf[128];  
  
    fd1 = open("/tmp/xyz", O_WRONLY | O_CREAT, S_IRUSR|S_IWUSR);  
    write(fd1, "hello", 5);  
    fd2 = open("/tmp/xyz", O_WRONLY, S_IRUSR|S_IWUSR);  
    write(fd2, "bye", 3);  
    close(fd1);  
    close(fd2);  
    return 0;  
}
```

Answer: byelo\0\0



The correct answer is: byelo

Question 3

Partially correct

Mark 0.60 out of 1.00

Select all the correct statements about two modes of CPU operation

Select one or more:

- a. Some instructions are allowed to run only in user mode, while all instructions can run in kernel mode
- b. There is an instruction like 'iret' to return from kernel mode to user mode✓
- c. The two modes are essential for a multitasking system✓
- d. The two modes are essential for a multiprogramming system✓
- e. The software interrupt instructions change the mode from user mode to kernel mode and jumps to predefined location simultaneously

Your answer is partially correct.

You have correctly selected 3.

The correct answers are: The two modes are essential for a multiprogramming system, The two modes are essential for a multitasking system, There is an instruction like 'iret' to return from kernel mode to user mode, The software interrupt instructions change the mode from user mode to kernel mode and jumps to predefined location simultaneously, Some instructions are allowed to run only in user mode, while all instructions can run in kernel mode

Question 4

Correct

Mark 0.50 out of 0.50

Compare multiprogramming with multitasking

- a. A multitasking system is not necessarily multiprogramming
- b. A multiprogramming system is not necessarily multitasking✓

The correct answer is: A multiprogramming system is not necessarily multitasking

Question 5

Partially correct

Mark 0.25 out of 1.00

Given below is the output of "ps -ef".

Answer the questions based on it.

UID	PID	PPID	C	S	TIME	STIME	TTY	CMD
root	1	0	0	Jan05	?	00:01:08	/sbin/init	splash
root	2	0	0	Jan05	?	00:00:00	[kthreadd]	
root	3	2	0	Jan05	?	00:00:00	[rcu_gp]	
root	4	2	0	Jan05	?	00:00:00	[rcu_par_gp]	
root	9	2	0	Jan05	?	00:00:00	[mm_percpu_wq]	
root	10	2	0	Jan05	?	00:00:00	[rcu_tasks_rude_]	
root	11	2	0	Jan05	?	00:00:00	[rcu_tasks_trace]	
root	12	2	0	Jan05	?	00:00:22	[ksoftirqd/0]	
root	13	2	0	Jan05	?	00:06:29	[rcu_sched]	
root	14	2	0	Jan05	?	00:00:02	[migration/0]	
root	15	2	0	Jan05	?	00:00:00	[idle_inject/0]	
root	16	2	0	Jan05	?	00:00:00	[cpuhp/0]	
root	17	2	0	Jan05	?	00:00:00	[cpuhp/1]	
root	18	2	0	Jan05	?	00:00:00	[idle_inject/1]	
root	19	2	0	Jan05	?	00:00:03	[migration/1]	
root	20	2	0	Jan05	?	00:00:13	[ksoftirqd/1]	
root	22	2	0	Jan05	?	00:00:00	[kworker/1:0H-events_highpri]	
root	23	2	0	Jan05	?	00:00:00	[cpuhp/2]	
root	24	2	0	Jan05	?	00:00:00	[idle_inject/2]	
root	25	2	0	Jan05	?	00:00:01	[migration/2]	
root	26	2	0	Jan05	?	00:00:09	[ksoftirqd/2]	
root	28	2	0	Jan05	?	00:00:00	[kworker/2:0H-kblockd]	
root	29	2	0	Jan05	?	00:00:00	[cpuhp/3]	
root	30	2	0	Jan05	?	00:00:00	[idle_inject/3]	
root	31	2	0	Jan05	?	00:00:02	[migration/3]	
root	32	2	0	Jan05	?	00:00:07	[ksoftirqd/3]	
root	34	2	0	Jan05	?	00:00:00	[kworker/3:0H-events_highpri]	
root	35	2	0	Jan05	?	00:00:00	[cpuhp/4]	
root	36	2	0	Jan05	?	00:00:00	[idle_inject/4]	
root	37	2	0	Jan05	?	00:00:02	[migration/4]	
root	38	2	0	Jan05	?	00:00:06	[ksoftirqd/4]	
root	40	2	0	Jan05	?	00:00:00	[kworker/4:0H-events_highpri]	
root	41	2	0	Jan05	?	00:00:00	[cpuhp/5]	
root	42	2	0	Jan05	?	00:00:00	[idle_inject/5]	
root	43	2	0	Jan05	?	00:00:02	[migration/5]	
root	44	2	0	Jan05	?	00:00:05	[ksoftirqd/5]	
root	46	2	0	Jan05	?	00:00:00	[kworker/5:0H-events_highpri]	
root	47	2	0	Jan05	?	00:00:00	[cpuhp/6]	
root	48	2	0	Jan05	?	00:00:00	[idle_inject/6]	
root	49	2	0	Jan05	?	00:00:02	[migration/6]	
root	50	2	0	Jan05	?	00:00:05	[ksoftirqd/6]	
root	52	2	0	Jan05	?	00:00:00	[kworker/6:0H-events_highpri]	
root	53	2	0	Jan05	?	00:00:00	[cpuhp/7]	
root	54	2	0	Jan05	?	00:00:00	[idle_inject/7]	
root	55	2	0	Jan05	?	00:00:02	[migration/7]	
root	56	2	0	Jan05	?	00:00:06	[ksoftirqd/7]	
root	58	2	0	Jan05	?	00:00:00	[kworker/7:0H-events_highpri]	
root	59	2	0	Jan05	?	00:00:00	[kdevtmpfs]	
root	60	2	0	Jan05	?	00:00:00	[netns]	
root	61	2	0	Jan05	?	00:00:00	[inet_frag_wq]	
root	62	2	0	Jan05	?	00:00:00	[kaudittd]	
root	63	2	0	Jan05	?	00:00:00	[khungtaskd]	
root	64	2	0	Jan05	?	00:00:00	[oom_reaper]	
root	65	2	0	Jan05	?	00:00:00	[writeback]	
root	66	2	0	Jan05	?	00:01:58	[kcompactd0]	
root	67	2	0	Jan05	?	00:00:00	[ksmd]	
root	68	2	0	Jan05	?	00:00:04	[khugepaged]	
root	115	2	0	Jan05	?	00:00:00	[kintegrityd]	
root	116	2	0	Jan05	?	00:00:00	[kblockd]	
root	117	2	0	Jan05	?	00:00:00	[blkcg_punt_bio]	
root	118	2	0	Jan05	?	00:00:00	[tpm_dev_wq]	

root 119 2 0 Jan05 ? 00:00:00 [ata_sff]
root 120 2 0 Jan05 ? 00:00:00 [md]
root 121 2 0 Jan05 ? 00:00:00 [edac-poller]
root 122 2 0 Jan05 ? 00:00:00 [devfreq_wq]
root 123 2 0 Jan05 ? 00:00:00 [watchdogd]
root 129 2 0 Jan05 ? 00:00:00 [irq/25-AMD-Vi]
root 131 2 0 Jan05 ? 00:04:33 [kswapd0]
root 132 2 0 Jan05 ? 00:00:00 [ecryptfs-kthrea]
root 134 2 0 Jan05 ? 00:00:00 [kthrotld]
root 135 2 0 Jan05 ? 00:00:00 [irq/27-pciehp]
root 139 2 0 Jan05 ? 00:00:00 [acpi_thermal_pm]
root 140 2 0 Jan05 ? 00:00:00 [vfio-irqfd-clea]
root 141 2 0 Jan05 ? 00:00:00 [ipv6_addrconf]
root 144 2 0 Jan05 ? 00:00:03 [kworker/6:1H-kblockd]
root 151 2 0 Jan05 ? 00:00:00 [kstrp]
root 154 2 0 Jan05 ? 00:00:00 [zswap-shrink]
root 162 2 0 Jan05 ? 00:00:00 [charger_manager]
root 164 2 0 Jan05 ? 00:00:03 [kworker/4:1H-kblockd]
root 197 2 0 Jan05 ? 00:00:03 [kworker/3:1H-kblockd]
root 213 2 0 Jan05 ? 00:00:03 [kworker/7:1H-kblockd]
root 215 2 0 Jan05 ? 00:00:00 [nvme-wq]
root 216 2 0 Jan05 ? 00:00:00 [nvme-reset-wq]
root 217 2 0 Jan05 ? 00:00:00 [nvme-delete-wq]
root 223 2 0 Jan05 ? 00:00:00 [irq/42-ELAN2513]
root 224 2 0 Jan05 ? 00:04:20 [irq/41-ELAN071B]
root 225 2 0 Jan05 ? 00:00:00 [cryptd]
root 226 2 0 Jan05 ? 00:00:00 [amd_iommu_v2]
root 249 2 0 Jan05 ? 00:00:00 [ttm_swap]
root 250 2 0 Jan05 ? 00:20:29 [gfx]
root 251 2 0 Jan05 ? 00:00:00 [comp_1.0.0]
root 252 2 0 Jan05 ? 00:00:00 [comp_1.1.0]
root 253 2 0 Jan05 ? 00:00:00 [comp_1.2.0]
root 254 2 0 Jan05 ? 00:00:00 [comp_1.3.0]
root 255 2 0 Jan05 ? 00:00:00 [comp_1.0.1]
root 256 2 0 Jan05 ? 00:00:00 [comp_1.1.1]
root 257 2 0 Jan05 ? 00:00:00 [comp_1.2.1]
root 258 2 0 Jan05 ? 00:00:00 [comp_1.3.1]
root 259 2 0 Jan05 ? 00:00:27 [sdma0]
root 260 2 0 Jan05 ? 00:00:00 [vcn_dec]
root 261 2 0 Jan05 ? 00:00:00 [vcn_enc0]
root 262 2 0 Jan05 ? 00:00:00 [vcn_enc1]
root 263 2 0 Jan05 ? 00:00:00 [jpeg_dec]
root 265 2 0 Jan05 ? 00:00:00 [card0-crtc0]
root 266 2 0 Jan05 ? 00:00:00 [card0-crtc1]
root 267 2 0 Jan05 ? 00:00:00 [card0-crtc2]
root 268 2 0 Jan05 ? 00:00:00 [card0-crtc3]
root 271 2 0 Jan05 ? 00:00:03 [kworker/5:1H-kblockd]
root 277 2 0 Jan05 ? 00:00:03 [kworker/1:1H-kblockd]
root 320 2 0 Jan05 ? 00:00:00 [raid5wq]
root 380 2 0 Jan05 ? 00:00:11 [jbd2/nvme0n1p5-]
root 381 2 0 Jan05 ? 00:00:00 [ext4-rsv-conver]
root 432 2 0 Jan05 ? 00:00:03 [kworker/2:1H-kblockd]
root 446 1 0 Jan05 ? 00:01:16 /lib/systemd/systemd-journald
root 470 2 0 Jan05 ? 00:00:00 [rpciod]
root 473 2 0 Jan05 ? 00:00:00 [xpriod]
root 474 2 0 Jan05 ? 00:00:00 bpfILTER_umh
root 503 1 0 Jan05 ? 00:00:07 /lib/systemd/systemd-udevd
root 517 2 0 Jan05 ? 00:00:00 [loop0]
root 554 2 0 Jan05 ? 00:00:00 [loop1]
root 563 2 0 Jan05 ? 00:00:00 [loop2]
root 586 2 0 Jan05 ? 00:00:00 [loop3]
root 588 2 0 Jan05 ? 00:00:00 [loop4]
root 589 2 0 Jan05 ? 00:00:00 [loop5]
root 612 2 0 Jan05 ? 00:00:00 [loop6]
root 613 2 0 Jan05 ? 00:00:00 [loop7]
root 629 2 0 Jan05 ? 00:00:00 [loop8]
root 637 2 0 Jan05 ? 00:00:00 [cfg80211]
root 676 2 0 Jan05 ? 00:05:00 [irq/75-iwlwifi:]
root 678 2 0 Jan05 ? 00:01:07 [irq/76-iwlwifi:]
root 682 2 0 Jan05 ? 00:01:27 [irq/77-iwlwifi:]

root 688 2 0 Jan05 ? 00:00:49 [irq/78-iwlwifi:]
root 695 2 0 Jan05 ? 00:01:39 [irq/79-iwlwifi:]
root 700 2 0 Jan05 ? 00:01:22 [irq/80-iwlwifi:]
root 703 2 0 Jan05 ? 00:01:13 [irq/81-iwlwifi:]
root 704 2 0 Jan05 ? 00:01:38 [irq/82-iwlwifi:]
root 708 2 0 Jan05 ? 00:00:44 [irq/83-iwlwifi:]
root 713 2 0 Jan05 ? 00:00:00 [loop9]
root 715 2 0 Jan05 ? 00:00:00 [irq/84-iwlwifi:]
root 782 2 0 Jan05 ? 00:00:00 [loop10]
root 797 2 0 Jan05 ? 00:00:00 [loop11]
root 811 2 0 Jan05 ? 00:00:00 [loop12]
root 838 2 0 Jan05 ? 00:00:00 [loop13]
root 847 2 0 Jan05 ? 00:00:00 [loop14]
root 879 2 0 Jan05 ? 00:00:00 [loop15]
root 884 2 0 Jan05 ? 00:00:00 [loop16]
root 885 2 0 Jan05 ? 00:00:00 [loop17]
root 945 2 0 Jan05 ? 00:00:00 [loop18]
root 946 2 0 Jan05 ? 00:00:00 [loop19]
root 947 2 0 Jan05 ? 00:00:00 [loop20]
root 1012 2 0 Jan05 ? 00:00:00 [jbd2/nvme0n1p8-]
root 1013 2 0 Jan05 ? 00:00:00 [ext4-rsv-conver]
root 1015 2 0 Jan05 ? 00:01:09 [jbd2/nvme0n1p7-]
root 1016 2 0 Jan05 ? 00:00:00 [ext4-rsv-conver]
_rpc 1062 1 0 Jan05 ? 00:00:00 /sbin/rpcbind -f -w
systemd+ 1063 1 0 Jan05 ? 00:01:24 /lib/systemd/systemd-resolved
systemd+ 1064 1 0 Jan05 ? 00:00:00 /lib/systemd/systemd-timesyncd
root 1144 1 0 Jan05 ? 00:00:46 /usr/sbin/acpid
avahi 1146 1 0 Jan05 ? 00:00:06 avahi-daemon: running [abhijit-laptop.local]
root 1149 1 0 Jan05 ? 00:00:01 /usr/lib/bluetooth/bluetoothd
message+ 1150 1 0 Jan05 ? 00:04:21 /usr/bin/dbus-daemon --system --address=systemd: --nofork --
nopidfile --systemd-activation --syslog-only
root 1152 1 0 Jan05 ? 00:03:12 /usr/sbin/NetworkManager -no-daemon
root 1157 1 0 Jan05 ? 00:01:02 /usr/sbin/iio-sensor-proxy
root 1159 1 0 Jan05 ? 00:00:27 /usr/sbin/irqbalance --foreground
root 1162 1 0 Jan05 ? 00:00:01 /usr/bin/lxcs /var/lib/lxcs
root 1165 1 0 Jan05 ? 00:00:00 /usr/bin/python3 /usr/bin/networkd-dispatcher --run-startup-
triggers
root 1170 1 0 Jan05 ? 00:00:29 /usr/lib/polkit-1/polkitd --no-debug
syslog 1175 1 0 Jan05 ? 00:00:20 /usr/sbin/rsyslogd -n -iNONE
root 1182 1 0 Jan05 ? 00:01:13 /usr/lib/snapd/snapd
root 1187 1 0 Jan05 ? 00:00:12 /usr/lib/accountsservice/accounts-daemon
root 1192 1 0 Jan05 ? 00:00:00 /usr/sbin/cron -f
root 1198 1 0 Jan05 ? 00:00:00 /usr/libexec/switcheroo-control
root 1201 1 0 Jan05 ? 00:00:12 /lib/systemd/systemd-logind
root 1202 1 0 Jan05 ? 00:00:07 /lib/systemd/systemd-machined
root 1203 1 0 Jan05 ? 00:01:28 /usr/lib/udisks2/udisksd
root 1204 1 0 Jan05 ? 00:00:15 /sbin/wpa_supplicant -u -s -0 /run/wpa_supplicant
daemon 1209 1 0 Jan05 ? 00:00:00 /usr/sbin/atd -f
avahi 1216 1146 0 Jan05 ? 00:00:00 avahi-daemon: chroot helper
docker-+ 1279 1 0 Jan05 ? 00:00:22 /usr/bin/docker-registry serve /etc/docker/registry/config.yml
root 1282 1 0 Jan05 ? 00:00:00 /usr/bin/python3 /usr/bin/twistd3 --nodaemon --pidfile= epoptes
jenkins 1285 1 0 Jan05 ? 00:15:01 /usr/bin/java -Djava.awt.headless=true -jar
/usr/share/java/jenkins.war --webroot=/var/cache/jenkins/war --httpPort=8080
root 1296 1 0 Jan05 ? 00:00:18 php-fpm: master process (/etc/php/7.4/fpm/php-fpm.conf)
vnstat 1314 1 0 Jan05 ? 00:00:15 /usr/sbin/vnstatd -n
root 1326 1 0 Jan05 ? 00:00:02 /usr/sbin/ModemManager
root 1327 1 0 Jan05 ? 00:00:31 /usr/bin/anydesk --service
colord 1359 1 0 Jan05 ? 00:00:01 /usr/libexec/colord
root 1374 1 0 Jan05 ? 00:00:00 /usr/sbin/gdm3
root 1420 1 0 Jan05 ? 00:00:14 /usr/sbin/apache2 -k start
www-data 1436 1296 0 Jan05 ? 00:00:00 php-fpm: pool www
www-data 1437 1296 0 Jan05 ? 00:00:00 php-fpm: pool www
mysql 1461 1 0 Jan05 ? 00:38:52 /usr/sbin/mysqld
root 1490 1 0 Jan05 ? 00:00:04 /usr/sbin/libvirtd
root 1491 1 0 Jan05 ? 00:00:00 /usr/bin/python3 /usr/share/unattended-upgrades/unattended-upgrade-
shutdown --wait-for-signal
rtkit 1593 1 0 Jan05 ? 00:00:07 /usr/libexec/rtkit-daemon
libvirt+ 1766 1 0 Jan05 ? 00:00:00 /usr/sbin/dnsmasq --conf-file=/var/lib/libvirt/dnsmasq/default.conf
--leasefile-ro --dhcp-script=/usr/lib/libvirt/libvirt_leaseshelper
root 1767 1766 0 Jan05 ? 00:00:00 /usr/sbin/dnsmasq --conf-file=/var/lib/libvirt/dnsmasq/default.conf

```
--leasefile-ro --dhcp-script=/usr/lib/libvirt/libvirt_leaseshelper
root      1859      1  0 Jan05 ?          00:00:18 /usr/lib/upower/upowerd
root      1995      1  0 Jan05 ?          00:00:00 /opt/saltstack/salt/run/run minion
root      2041    1995  0 Jan05 ?          00:05:18 /opt/saltstack/salt/run/run minion MultiMinionProcessManager
MinionProcessManager
root      2278      1  0 Jan05 ?          00:00:50 /usr/bin/dockerd -H fd:// --
containerd=/run/containerd/containerd.sock
root      2282      1  0 Jan05 ?          00:00:17 /usr/sbin/inetd
whoopsie  2302      1  0 Jan05 ?          00:00:01 /usr/bin/whoopsie -f
kernoops  2330      1  0 Jan05 ?          00:00:19 /usr/sbin/kerneloops --test
kernoops  2341      1  0 Jan05 ?          00:00:19 /usr/sbin/kerneloops
root      2366      2  0 Jan05 ?          00:00:00 [iprt-VBoxWQueue]
lxc-dns+  2370      1  0 Jan05 ?          00:00:00 dnsmasq --conf-file=/dev/null -u lxc-dnsmasq --strict-order --bind-
interfaces --pid-file=/run/lxc/dnsmasq.pid --listen-address 10.0.3.1 --dhcp-range 10.0.3.2,10.0.3.254 --dhcp-lease-
max=253 --dhcp-no-override --except-interface=lo --interface=lxcbr0 --dhcp-
leasefile=/var/lib/misc/dnsmasq.lxcbr0.leases --dhcp-authoritative
root      2404      2  0 Jan05 ?          00:00:00 [iprt-VBoxTscThr]
root      3508      1  0 Jan05 ?          00:00:01 /usr/lib/postfix/sbin/master -w
root      3615    1374  0 Jan05 ?          00:00:01 gdm-session-worker [pam/gdm-password]
abhijit   3629      1  0 Jan05 ?          00:00:17 /lib/systemd/systemd --user
abhijit   3630    3629  0 Jan05 ?          00:00:00 (sd-pam)
abhijit   3636    3629  1 Jan05 ?          04:31:21 /usr/bin/pulseaudio --daemonize=no --log-target=journal
abhijit   3638    3629  0 Jan05 ?          00:20:35 /usr/libexec/tracker-miner-fs
abhijit   3642    3629  0 Jan05 ?          00:01:11 /usr/bin/dbus-daemon --session --address=systemd: --nofork --
nopidfile --systemd-activation --syslog-only
abhijit   3644      1  0 Jan05 ?          00:00:03 /usr/bin/gnome-keyring-daemon --daemonize --login
abhijit   3662    3629  0 Jan05 ?          00:00:01 /usr/libexec/gvfsd
abhijit   3667    3629  0 Jan05 ?          00:00:00 /usr/libexec/gvfsd-fuse /run/user/1000/gvfs -f -o big_writes
abhijit   3673    3629  0 Jan05 ?          00:00:02 /usr/libexec/gvfs-udisks2-volume-monitor
abhijit   3681    3629  0 Jan05 ?          00:00:01 /usr/libexec/gvfs-mtp-volume-monitor
abhijit   3685    3629  0 Jan05 ?          00:00:11 /usr/libexec/gvfs-afc-volume-monitor
abhijit   3690    3629  0 Jan05 ?          00:00:01 /usr/libexec/gvfs-gphoto2-volume-monitor
abhijit   3695    3629  0 Jan05 ?          00:00:01 /usr/libexec/gvfs-goa-volume-monitor
abhijit   3700    3629  0 Jan05 ?          00:00:01 /usr/libexec/goa-daemon
root      3701      2  0 Jan05 ?          00:00:00 [krfcomm]
abhijit   3708    3629  0 Jan05 ?          00:00:04 /usr/libexec/goa-identity-service
abhijit   3724    3615  tty2      00:00:00 /usr/lib/gdm3/gdm-x-session --run-script env
GNOME_SHELL_SESSION_MODE=ubuntu /usr/bin/gnome-session --systemd --session=ubuntu
abhijit   3726    3724  1 Jan05 tty2      02:47:57 /usr/lib/xorg/Xorg vt2 -displayfd 3 -auth
/run/user/1000/gdm/Xauthority -background none -noreset -keeptty -verbose 3
abhijit   3747    3724  0 Jan05 tty2      00:00:00 /usr/libexec/gnome-session-binary --systemd --systemd --
session=ubuntu
abhijit   3816    3747  0 Jan05 ?          00:00:01 /usr/bin/ssh-agent /usr/bin/im-launch env
GNOME_SHELL_SESSION_MODE=ubuntu /usr/bin/gnome-session --systemd --session=ubuntu
abhijit   3845    3629  0 Jan05 ?          00:00:00 /usr/libexec/at-spi-bus-launcher
abhijit   3850    3845  0 Jan05 ?          00:00:07 /usr/bin/dbus-daemon --config-file=/usr/share/defaults/at-
spi2/accessibility.conf --nofork --print-address 3
abhijit   3869    3629  0 Jan05 ?          00:00:00 /usr/libexec/gnome-session-ctl --monitor
abhijit   3876    3629  0 Jan05 ?          00:00:04 /usr/libexec/gnome-session-binary --systemd-service --
session=ubuntu
abhijit   3890    3629  1 Jan05 ?          03:43:47 /usr/bin/gnome-shell
abhijit   3927    3890  0 Jan05 ?          00:31:49 ibus-daemon --panel disable --xim
abhijit   3931    3927  0 Jan05 ?          00:00:00 /usr/libexec/ibus-dconf
abhijit   3932    3927  0 Jan05 ?          00:01:51 /usr/libexec/ibus-extension-gtk3
abhijit   3934    3629  0 Jan05 ?          00:00:04 /usr/libexec/ibus-x11 --kill-daemon
abhijit   3937    3629  0 Jan05 ?          00:00:02 /usr/libexec/ibus-portal
abhijit   3949    3629  0 Jan05 ?          00:00:27 /usr/libexec/at-spi2-registryd --use-gnome-session
abhijit   3953    3629  0 Jan05 ?          00:00:00 /usr/libexec/xdg-permission-store
abhijit   3955    3629  0 Jan05 ?          00:00:01 /usr/libexec/gnome-shell-calendar-server
abhijit   3964    3629  0 Jan05 ?          00:00:00 /usr/libexec/evolution-source-registry
abhijit   3973    3629  0 Jan05 ?          00:00:02 /usr/libexec/evolution-calendar-factory
abhijit   3986    3629  0 Jan05 ?          00:00:01 /usr/libexec/dconf-service
abhijit   3992    3629  0 Jan05 ?          00:00:01 /usr/libexec/evolution-addressbook-factory
abhijit   4007    3629  0 Jan05 ?          00:00:00 /usr/bin/gjs /usr/share/gnome-shell/org.gnome.Shell.Notifications
abhijit   4023    3629  0 Jan05 ?          00:00:00 /usr/libexec/gsd-a11y-settings
abhijit   4025    3629  0 Jan05 ?          00:00:08 /usr/libexec/gsd-color
abhijit   4029    3629  0 Jan05 ?          00:00:00 /usr/libexec/gsd-datetime
abhijit   4032    3629  0 Jan05 ?          00:00:21 /usr/libexec/gsd-housekeeping
abhijit   4033    3629  0 Jan05 ?          00:00:05 /usr/libexec/gsd-keyboard
abhijit   4036    3629  0 Jan05 ?          00:00:12 /usr/libexec/gsd-media-keys
```

abhijit 4037 3629 0 Jan05 ? 00:00:11 /usr/libexec/gsd-power
abhijit 4038 3629 0 Jan05 ? 00:00:00 /usr/libexec/gsd-print-notifications
abhijit 4039 3629 0 Jan05 ? 00:00:01 /usr/libexec/gsd-rfkill
abhijit 4041 3629 0 Jan05 ? 00:00:01 /usr/libexec/gsd-screensaver-proxy
abhijit 4042 3876 0 Jan05 ? 00:01:06 /usr/lib/x86_64-linux-gnu/libexec/kdeconnectd
abhijit 4045 3629 0 Jan05 ? 00:00:35 /usr/libexec/gsd-sharing
abhijit 4047 3629 0 Jan05 ? 00:00:00 /usr/libexec/gsd-smartcard
abhijit 4051 3629 0 Jan05 ? 00:00:00 /usr/libexec/gsd-sound
abhijit 4057 3629 0 Jan05 ? 00:00:00 /usr/libexec/gsd-usb-protection
abhijit 4063 3629 0 Jan05 ? 00:00:05 /usr/libexec/gsd-wacom
abhijit 4071 3629 0 Jan05 ? 00:00:00 /usr/libexec/gsd-wwan
abhijit 4072 3876 0 Jan05 ? 00:01:00 baloo_file
abhijit 4075 3876 0 Jan05 ? 00:00:00 /usr/libexec/gsd-disk-utility-notify
abhijit 4076 3629 0 Jan05 ? 00:00:08 /usr/libexec/gsd-xsettings
abhijit 4078 3876 0 Jan05 ? 00:00:10 /usr/bin/python3 /usr/bin/blueman-applet
abhijit 4082 3876 0 Jan05 ? 00:00:14 /usr/bin/anydesk --tray
abhijit 4108 3876 0 Jan05 ? 00:00:00 /usr/lib/x86_64-linux-gnu/indicator-messages/indicator-messages-service
abhijit 4109 3876 0 Jan05 ? 00:00:06 /usr/libexec/evolution-data-server/evolution-alarm-notify
abhijit 4129 3629 0 Jan05 ? 00:00:50 /snap/snap-store/959/usr/bin/snap-store --gapplication-service
abhijit 4191 3629 0 Jan05 ? 00:00:00 /usr/libexec/gsd-printer
abhijit 4219 3629 0 Jan05 ? 00:00:03 /usr/libexec/xdg-document-portal
abhijit 4265 3927 0 Jan05 ? 00:03:20 /usr/libexec/ibus-engine-simple
abhijit 4291 3629 0 Jan05 ? 00:00:10 /usr/bin/python3 /usr/bin/blueman-tray
abhijit 4301 3629 0 Jan05 ? 00:00:00 /usr/lib/bluetooth/obexd
abhijit 4377 3662 0 Jan05 ? 00:00:02 /usr/libexec/gvfsd-trash --spawner :1.3 /org/gtk/gvfs/exec_spaw/0
abhijit 4395 3629 0 Jan05 ? 00:00:12 /usr/libexec/xdg-desktop-portal
abhijit 4399 3629 0 Jan05 ? 00:01:08 /usr/libexec/xdg-desktop-portal-gtk
abhijit 4480 3629 0 Jan05 ? 00:00:21 /usr/libexec/gvfsd-metadata
abhijit 5687 3662 0 Jan05 ? 00:00:00 /usr/libexec/gvfsd-network --spawner :1.3 /org/gtk/gvfs/exec_spaw/1
abhijit 5701 3662 0 Jan05 ? 00:00:01 /usr/libexec/gvfsd-dnssd --spawner :1.3 /org/gtk/gvfs/exec_spaw/3
root 6228 2 0 Jan05 ? 00:00:00 [kdmflush]
root 6236 2 0 Jan05 ? 00:00:00 [kcryptd_io/253:]
root 6237 2 0 Jan05 ? 00:00:00 [kcryptd/253:0]
root 6238 2 0 Jan05 ? 00:00:23 [dmcrypt_write/2]
root 6260 2 0 Jan05 ? 00:00:24 [jbd2/dm-0-8]
root 6261 2 0 Jan05 ? 00:00:00 [ext4-rsv-conver]
abhijit 6421 3927 0 Jan05 ? 00:00:36 /usr/lib/ibus/ibus-engine-m17n --ibus
abhijit 6434 3876 0 Jan05 ? 00:00:13 update-notifier
abhijit 30565 3629 0 Jan05 ? 00:08:56 /usr/libexec/gnome-terminal-server
abhijit 30576 30565 0 Jan05 pts/0 00:00:00 bash
abhijit 131017 364845 1 Jan09 ? 03:12:27 /usr/lib/virtualbox/VirtualBoxVM --comment ubuntu 18.04 --startvm 45993a5c-3ded-452f-941e-4579d12c1ad9 --no-startvm-errormsgbox
abhijit 159668 30565 0 Jan09 pts/10 00:00:00 bash
abhijit 161637 30565 0 Jan05 pts/1 00:00:01 bash
abhijit 171109 159668 0 Jan09 pts/10 00:00:33 evince.....
abhijit 171114 3629 0 Jan09 ? 00:00:00 /usr/libexec/evinced
abhijit 204055 3629 0 Jan10 ? 00:00:13 /usr/bin/python3 /usr/bin/update-manager --no-update --no-focus-on-map
abhijit 270190 3629 0 Jan05 ? 00:56:34 /usr/lib/x86_64-linux-gnu/libexec/kactivitymanagerd
abhijit 270199 3629 0 Jan05 ? 00:00:24 /usr/bin/kglobalaccel5
abhijit 270207 3629 0 Jan05 ? 00:00:00 kdeinit5: Running...
abhijit 270208 270207 0 Jan05 ? 00:00:19 /usr/lib/x86_64-linux-gnu/libexec/kf5/klauncher --fd=8
abhijit 279971 3629 0 Jan05 ? 01:20:12 telegram-desktop
abhijit 280011 279971 0 Jan05 ? 00:00:00 sh -c /usr/lib/x86_64-linux-gnu/libproxy/0.4.15/pxgsettings
org.gnome.system.proxy org.gnome.system.proxy.http org.gnome.system.proxy.https org.gnome.system.proxy.ftp
org.gnome.system.proxy.socks
abhijit 280015 280011 0 Jan05 ? 00:00:00 /usr/lib/x86_64-linux-gnu/libproxy/0.4.15/pxgsettings
org.gnome.system.proxy org.gnome.system.proxy.http org.gnome.system.proxy.https org.gnome.system.proxy.ftp
org.gnome.system.proxy.socks
abhijit 325756 3629 0 Jan12 ? 00:01:09 /usr/libexec/tracker-store
root 326592 1 0 Jan12 ? 00:00:00 sshd: /usr/sbin/sshd -D [listener] 0 of 10-100 startups
abhijit 347912 3644 0 Jan06 ? 00:00:00 /usr/bin/ssh-agent -D -a /run/user/1000/keyring/.ssh
root 348716 1 0 Jan12 ? 00:00:28 /usr/bin/containerd
abhijit 351306 3629 3 Jan12 ? 03:32:42 /usr/lib/firefox/firefox
abhijit 351429 351306 0 Jan12 ? 00:00:00 /usr/lib/firefox/firefox -contentproc -parentBuildID 20240108143603
-prefsLen 37272 -prefMapSize 247458 -appDir /usr/lib/firefox/browser {83364ade-74ec-4bcc-94f8-9f3779a869f1} 351306 true
socket
abhijit 351458 351306 0 Jan12 ? 00:29:34 /usr/lib/firefox/firefox -contentproc -childID 1 -isForBrowser -
prefsLen 37337 -prefMapSize 247458 -jsInitLen 229864 -parentBuildID 20240108143603 -greomni /usr/lib/firefox/omni.ja -

appomni /usr/lib/firefox/browser/omni.ja -appDir /usr/lib/firefox/browser {4dbb54ee-e1a2-41e4-b10b-587526532b78} 351306
true tab
abhijit 351495 351306 0 Jan12 ? 00:04:06 /usr/lib/firefox/firefox -contentproc -childID 2 -isForBrowser -
prefsLen 38090 -prefMapSize 247458 -jsInitLen 229864 -parentBuildID 20240108143603 -greomni /usr/lib/firefox/omni.ja -
appomni /usr/lib/firefox/browser/omni.ja -appDir /usr/lib/firefox/browser {02f1a4e7-d831-4a5f-a9fd-59a809bb03e8} 351306
true tab
abhijit 351717 351306 0 Jan12 ? 00:01:14 /usr/lib/firefox/firefox -contentproc -parentBuildID 20240108143603 -
-sandboxingKind 0 -prefsLen 42823 -prefMapSize 247458 -appDir /usr/lib/firefox/browser {a1e46009-cfd1-46c8-ac9c-
19ba8bf3f46a} 351306 true utility
abhijit 351844 351306 0 Jan12 ? 00:07:23 /usr/lib/firefox/firefox -contentproc -parentBuildID 20240108143603 -
-prefsLen 43306 -prefMapSize 247458 -appDir /usr/lib/firefox/browser {91008bef-b6d6-452f-b4a6-e78d887b3569} 351306 true
rdd
abhijit 353500 3662 0 Jan06 ? 00:00:00 /usr/libexec/gvfsd-http --spawner :1.3 /org/gtk/gvfs/exec_spaw/4
abhijit 364809 3890 0 Jan06 ? 00:19:20 /usr/lib/virtualbox/VirtualBox
abhijit 364837 3629 0 Jan06 ? 00:16:49 /usr/lib/virtualbox/VBoxXPComIPCD
abhijit 364845 3629 0 Jan06 ? 00:29:44 /usr/lib/virtualbox/VBoxSVC --auto-shutdown
root 364957 2 0 Jan06 ? 00:00:00 [dio/dm-0]
abhijit 369749 30565 0 Jan06 pts/3 00:00:00 bash
abhijit 369879 30565 0 Jan06 pts/4 00:00:00 bash
abhijit 379620 30565 0 Jan06 pts/6 00:00:00 bash
abhijit 381917 3890 0 Jan06 ? 00:00:46 flameshot
root 386201 2 0 Jan06 ? 00:00:02 [kworker/0:2H-acpi_thermal_pm]
postfix 389291 3508 0 Jan06 ? 00:00:00 qmgr -l -t unix -u
root 486171 2 0 Jan12 ? 00:00:01 [kworker/0:0H-kblockd]
abhijit 527395 3629 0 Jan12 ? 00:00:49 /usr/bin/gedit --gapplication-service
abhijit 551299 351306 0 Jan12 ? 00:01:54 /usr/lib/firefox/firefox -contentproc -childID 438 -isForBrowser -
prefsLen 35109 -prefMapSize 247458 -jsInitLen 229864 -parentBuildID 20240108143603 -greomni /usr/lib/firefox/omni.ja -
appomni /usr/lib/firefox/browser/omni.ja -appDir /usr/lib/firefox/browser {c4ebd70d-bad7-4e7c-9066-39827103c84e} 351306
true tab
abhijit 552002 3890 0 Jan12 ? 00:03:32 /opt/Signal/signal-desktop --no-sandbox
abhijit 552005 552002 0 Jan12 ? 00:00:00 /opt/Signal/signal-desktop --type=zygote --no-zygote-sandbox --no-
sandbox
abhijit 552006 552002 0 Jan12 ? 00:00:00 /opt/Signal/signal-desktop --type=zygote --no-sandbox
abhijit 552037 552005 0 Jan12 ? 00:03:28 /opt/Signal/signal-desktop --type=gpu-process --no-sandbox --
enable-crash-reporter=18887fa1-4d37-46f0-bf9f-b37513c81cf9,no_channel --user-data-dir=/home/abhijit/.config/Signal --
gpu-
preferences=WAAAAAAAAGAAAAEAAAAAAAAAAAAAAABgAAAAAAA4AAAAAAA4AAAAAAA4AAAAAAA4AAAAAAA4AAAAAAA4AAAAAABAAAAGAAAAAAAAYAAA
--use-gl=angle --use-angle=swiftshader-webgl --shared-files --field-trial-
handle=0,i,3213315393136307567,15155041764863120512,262144 --disable-
features=HardwareMediaKeyHandling,SpareRendererForSitePerProcess
abhijit 552045 552002 0 Jan12 ? 00:00:04 /opt/Signal/signal-desktop --type=utility --utility-sub-
type=network.mojom.NetworkService --lang=en-GB --service-sandbox-type=none --no-sandbox --enable-crash-
reporter=18887fa1-4d37-46f0-bf9f-b37513c81cf9,no_channel --user-data-dir=/home/abhijit/.config/Signal --shared-
files=v8_context_snapshot_data:100 --field-trial-handle=0,i,3213315393136307567,15155041764863120512,262144 --disable-
features=HardwareMediaKeyHandling,SpareRendererForSitePerProcess
abhijit 552103 552002 1 Jan12 ? 00:57:51 /opt/Signal/signal-desktop --type=renderer --enable-crash-
reporter=18887fa1-4d37-46f0-bf9f-b37513c81cf9,no_channel --user-data-dir=/home/abhijit/.config/Signal --app-
path=/opt/Signal/resources/app.asar --no-sandbox --no-zygote --enable-blink-features=CSSPseudoDir,CSSLogical --disable-
blink-features=Accelerated2dCanvas,AcceleratedSmallCanvases --first-renderer-process --no-sandbox --disable-gpu-
compositing --lang=en-GB --num-raster-threads=4 --enable-main-frame-before-activation --renderer-client-id=4 --time-
ticks-at-unix-epoch=-1704847690836396 --launch-time-ticks=218630770368 --shared-files=v8_context_snapshot_data:100 --
field-trial-handle=0,i,3213315393136307567,15155041764863120512,262144 --disable-
features=HardwareMediaKeyHandling,SpareRendererForSitePerProcess
abhijit 552149 552002 0 Jan12 ? 00:00:12 /opt/Signal/signal-desktop --type=utility --utility-sub-
type=audio.mojom.AudioService --lang=en-GB --service-sandbox-type=none --no-sandbox --enable-crash-reporter=18887fa1-
4d37-46f0-bf9f-b37513c81cf9,no_channel --user-data-dir=/home/abhijit/.config/Signal --shared-
files=v8_context_snapshot_data:100 --field-trial-handle=0,i,3213315393136307567,15155041764863120512,262144 --disable-
features=HardwareMediaKeyHandling,SpareRendererForSitePerProcess
abhijit 554660 351306 0 Jan13 ? 00:04:20 /usr/lib/firefox/firefox -contentproc -childID 442 -isForBrowser -
prefsLen 35109 -prefMapSize 247458 -jsInitLen 229864 -parentBuildID 20240108143603 -greomni /usr/lib/firefox/omni.ja -
appomni /usr/lib/firefox/browser/omni.ja -appDir /usr/lib/firefox/browser {2cffa6f3-6f8c-4a3f-880d-01cc71baf983} 351306
true tab
abhijit 554855 351306 4 Jan13 ? 03:24:06 /usr/lib/firefox/firefox -contentproc -childID 445 -isForBrowser -
prefsLen 35109 -prefMapSize 247458 -jsInitLen 229864 -parentBuildID 20240108143603 -greomni /usr/lib/firefox/omni.ja -
appomni /usr/lib/firefox/browser/omni.ja -appDir /usr/lib/firefox/browser {abcba8e9-5b97-4281-ad64-1356841dcf34} 351306
true tab
abhijit 556349 351306 0 Jan13 ? 00:01:18 /usr/lib/firefox/firefox -contentproc -childID 451 -isForBrowser -
prefsLen 35109 -prefMapSize 247458 -jsInitLen 229864 -parentBuildID 20240108143603 -greomni /usr/lib/firefox/omni.ja -
appomni /usr/lib/firefox/browser/omni.ja -appDir /usr/lib/firefox/browser {6318d453-643c-4b85-9f0f-bfd0a429cd41} 351306
true tab

abhijit 557600 3629 0 Jan13 ? 00:00:08 /usr/lib/speech-dispatcher-modules/sd_espeak-ng /etc/speech-dispatcher/modules/espeak-ng.conf
abhijit 557607 3629 0 Jan13 ? 00:00:08 /usr/lib/speech-dispatcher-modules/sd_dummy /etc/speech-dispatcher/modules/dummy.conf
abhijit 557610 3629 0 Jan13 ? 00:00:08 /usr/lib/speech-dispatcher-modules/sd_generic /etc/speech-dispatcher/modules/mary-generic.conf
abhijit 557613 3629 0 Jan13 ? 00:00:00 /usr/bin/speech-dispatcher --spawn --communication-method unix_socket --socket-path /run/user/1000/speech-dispatcher/speechd.sock
abhijit 571320 351306 0 Jan13 ? 00:01:16 /usr/lib/firefox/firefox -contentproc -childID 486 -isForBrowser -prefsLen 35110 -prefMapSize 247458 -jsInitLen 229864 -parentBuildID 20240108143603 -greomni /usr/lib/firefox/omni.ja -appomni /usr/lib/firefox/browser/omni.ja -appDir /usr/lib/firefox/browser {71966a59-b1df-4c44-975c-62d7cd41188c} 351306 true tab
abhijit 571410 351306 0 Jan13 ? 00:01:34 /usr/lib/firefox/firefox -contentproc -childID 488 -isForBrowser -prefsLen 35110 -prefMapSize 247458 -jsInitLen 229864 -parentBuildID 20240108143603 -greomni /usr/lib/firefox/omni.ja -appomni /usr/lib/firefox/browser/omni.ja -appDir /usr/lib/firefox/browser {bfbc15ff-6ca5-4837-b97e-334e0bfc8855} 351306 true tab
abhijit 571582 351306 0 Jan13 ? 00:03:32 /usr/lib/firefox/firefox -contentproc -childID 492 -isForBrowser -prefsLen 35110 -prefMapSize 247458 -jsInitLen 229864 -parentBuildID 20240108143603 -greomni /usr/lib/firefox/omni.ja -appomni /usr/lib/firefox/browser/omni.ja -appDir /usr/lib/firefox/browser {04e791e5-9cde-4e03-9211-141bdb440139} 351306 true tab
abhijit 591339 4139495 0 Jan13 pts/9 00:00:00 vi mh-pyt.txt
root 594356 2 0 Jan13 ? 00:00:00 [dio/nvme0n1p7]
abhijit 594726 30565 0 Jan13 pts/5 00:00:00 bash
abhijit 791421 351306 1 Jan14 ? 00:40:27 /usr/lib/firefox/firefox -contentproc -childID 839 -isForBrowser -prefsLen 35227 -prefMapSize 247458 -jsInitLen 229864 -parentBuildID 20240108143603 -greomni /usr/lib/firefox/omni.ja -appomni /usr/lib/firefox/browser/omni.ja -appDir /usr/lib/firefox/browser {b6f69be3-7ebe-43c9-9d55-c72161180b5e} 351306 true tab
abhijit 794226 351306 0 Jan14 ? 00:02:03 /usr/lib/firefox/firefox -contentproc -childID 848 -isForBrowser -prefsLen 35227 -prefMapSize 247458 -jsInitLen 229864 -parentBuildID 20240108143603 -greomni /usr/lib/firefox/omni.ja -appomni /usr/lib/firefox/browser/omni.ja -appDir /usr/lib/firefox/browser {28fddc81-ea39-4496-a1b6-b70d9a8a9c31} 351306 true tab
abhijit 797871 351306 0 Jan14 ? 00:02:41 /usr/lib/firefox/firefox -contentproc -childID 851 -isForBrowser -prefsLen 35226 -prefMapSize 247458 -jsInitLen 229864 -parentBuildID 20240108143603 -greomni /usr/lib/firefox/omni.ja -appomni /usr/lib/firefox/browser/omni.ja -appDir /usr/lib/firefox/browser {b254a3c7-e1f7-4245-9fcf-820074a4e69f} 351306 true tab
abhijit 799607 30565 0 Jan14 pts/11 00:00:00 bash
abhijit 803503 30565 0 Jan14 pts/12 00:00:00 bash
abhijit 815513 799607 0 Jan15 pts/11 00:00:00 vi timetable-todo2
abhijit 821738 351306 0 Jan15 ? 00:07:30 /usr/lib/firefox/firefox -contentproc -childID 995 -isForBrowser -prefsLen 35227 -prefMapSize 247458 -jsInitLen 229864 -parentBuildID 20240108143603 -greomni /usr/lib/firefox/omni.ja -appomni /usr/lib/firefox/browser/omni.ja -appDir /usr/lib/firefox/browser {90cad852-941e-44b3-8fa4-0d44cbcfda7a} 351306 true tab
abhijit 895193 351306 0 Jan15 ? 00:00:04 /usr/lib/firefox/firefox -contentproc -childID 1131 -isForBrowser -prefsLen 35227 -prefMapSize 247458 -jsInitLen 229864 -parentBuildID 20240108143603 -greomni /usr/lib/firefox/omni.ja -appomni /usr/lib/firefox/browser/omni.ja -appDir /usr/lib/firefox/browser {218262bd-4f9e-423a-9368-4d2e44f33125} 351306 true tab
root 942398 1 0 10:40 ? 00:00:00 /usr/sbin/cupsd -l
root 942399 1 0 10:40 ? 00:00:00 /usr/sbin/cups-browsed
www-data 942465 1420 0 10:40 ? 00:00:00 /usr/sbin/apache2 -k start
www-data 942466 1420 0 10:40 ? 00:00:00 /usr/sbin/apache2 -k start
www-data 942468 1420 0 10:40 ? 00:00:00 /usr/sbin/apache2 -k start
www-data 942469 1420 0 10:40 ? 00:00:00 /usr/sbin/apache2 -k start
www-data 942470 1420 0 10:40 ? 00:00:00 /usr/sbin/apache2 -k start
www-data 942471 1420 0 10:40 ? 00:00:00 /usr/sbin/apache2 -k start
abhijit 954109 30565 0 11:21 pts/2 00:00:00 bash
abhijit 961628 351306 3 12:36 ? 00:05:34 /usr/lib/firefox/firefox -contentproc -childID 1704 -isForBrowser -prefsLen 35227 -prefMapSize 247458 -jsInitLen 229864 -parentBuildID 20240108143603 -greomni /usr/lib/firefox/omni.ja -appomni /usr/lib/firefox/browser/omni.ja -appDir /usr/lib/firefox/browser {cb82b978-ef9d-4a76-b59c-5b13b652a0ec} 351306 true tab
root 961877 2 0 12:36 ? 00:00:02 [kworker/u32:5-events_unbound]
root 962113 2 0 12:39 ? 00:00:08 [kworker/u33:3-hci0]
abhijit 968060 351306 0 13:09 ? 00:00:05 /usr/lib/firefox/firefox -contentproc -childID 1749 -isForBrowser -prefsLen 35227 -prefMapSize 247458 -jsInitLen 229864 -parentBuildID 20240108143603 -greomni /usr/lib/firefox/omni.ja -appomni /usr/lib/firefox/browser/omni.ja -appDir /usr/lib/firefox/browser {c0ba3673-966e-48a6-b029-91346e348342} 351306 true tab
root 968299 2 0 13:11 ? 00:00:02 [kworker/u32:4-events_unbound]
root 969560 2 0 13:23 ? 00:00:01 [kworker/5:1-cgroup_destroy]
root 969608 2 0 13:24 ? 00:00:00 [kworker/u32:7-events_unbound]
root 969648 2 0 13:24 ? 00:00:01 [kworker/2:0-inet_frag_wq]
abhijit 969715 379620 0 13:25 pts/6 00:00:00 ssh root@10.1.101.41

root	970435	2 0 13:27 ?	00:00:01 [kworker/3:2-rcu_gp]
root	971921	2 0 13:33 ?	00:00:00 [kworker/1:2-rcu_gp]
root	972048	2 0 13:35 ?	00:00:00 [kworker/7:2-rcu_gp]
root	972127	2 0 13:37 ?	00:00:01 [kworker/0:1-events_long]
root	972207	2 0 13:37 ?	00:00:01 [kworker/4:1-events]
root	972378	2 0 13:39 ?	00:00:00 [kworker/1:0-cgroup_destroy]
root	972435	2 0 13:39 ?	00:00:00 [kworker/6:2-events]
root	972964	2 0 13:41 ?	00:00:00 [kworker/7:0-events]
root	973166	2 0 13:41 ?	00:00:00 [kworker/u32:0-events_unbound]
root	973193	2 0 13:42 ?	00:00:00 [kworker/2:1-events]
root	973282	2 0 13:43 ?	00:00:00 [kworker/4:2-events]
root	973384	2 0 13:44 ?	00:00:00 [kworker/3:0-rcu_gp]
root	973389	2 0 13:44 ?	00:00:01 [kworker/u33:0-hci0]
root	973391	2 0 13:44 ?	00:00:00 [kworker/6:0-rcu_gp]
root	973392	2 0 13:44 ?	00:00:00 [kworker/5:2-events]
root	973655	2 0 13:45 ?	00:00:00 [kworker/1:1-events]
root	973664	2 0 13:46 ?	00:00:00 [kworker/7:1-events]
root	973965	2 0 13:47 ?	00:00:00 [kworker/2:2-events]
root	974126	2 0 13:48 ?	00:00:00 [kworker/u32:1-kcryptd/253:0]
root	974189	2 0 13:48 ?	00:00:00 [kworker/4:0-events]
root	974190	2 0 13:48 ?	00:00:00 [kworker/0:2-events]
root	974369	2 0 13:49 ?	00:00:00 [kworker/u32:2-nvme-wq]
root	974539	2 0 13:49 ?	00:00:00 [kworker/u32:3-events_unbound]
root	974655	2 0 13:50 ?	00:00:00 [kworker/6:1-events]
root	974690	2 0 13:50 ?	00:00:00 [kworker/5:0-events]
root	974740	2 0 13:50 ?	00:00:00 [kworker/7:3-events]
root	974742	2 0 13:50 ?	00:00:00 [kworker/3:1-pm]
root	974743	2 0 13:50 ?	00:00:00 [kworker/u32:6-events_unbound]
root	974744	2 0 13:50 ?	00:00:00 [kworker/u32:8-kcryptd/253:0]
root	974745	2 0 13:50 ?	00:00:00 [kworker/u32:9-events_unbound]
root	974746	2 0 13:50 ?	00:00:00 [kworker/u32:10-events_unbound]
root	974747	2 0 13:50 ?	00:00:00 [kworker/u32:11-events_unbound]
root	974748	2 0 13:50 ?	00:00:00 [kworker/u32:12-kcryptd/253:0]
root	974749	2 0 13:50 ?	00:00:00 [kworker/u32:13-events_unbound]
root	974750	2 0 13:50 ?	00:00:00 [kworker/u32:14-events_unbound]
root	974751	2 0 13:50 ?	00:00:00 [kworker/u32:15-events_unbound]
root	974752	2 0 14:18 ?	00:00:00 [kworker/u32:16-events_unbound]
root	974753	2 0 14:18 ?	00:00:00 [kworker/u32:17-events_unbound]
root	974754	2 0 14:18 ?	00:00:00 [kworker/u32:18-events_unbound]
root	974755	2 0 14:18 ?	00:00:00 [kworker/u32:19-events_unbound]
root	974756	2 0 14:18 ?	00:00:00 [kworker/u32:20-kcryptd/253:0]
root	974757	2 0 14:18 ?	00:00:00 [kworker/u32:21-events_unbound]
root	974758	2 0 14:18 ?	00:00:00 [kworker/u32:22-events_unbound]
root	974759	2 0 14:18 ?	00:00:00 [kworker/u32:23-events_unbound]
root	974760	2 0 14:18 ?	00:00:00 [kworker/u32:24-events_unbound]
root	974761	2 0 14:18 ?	00:00:00 [kworker/u32:25-events_unbound]
root	974762	2 0 14:18 ?	00:00:00 [kworker/u32:26-events_unbound]
root	974763	2 0 14:18 ?	00:00:00 [kworker/u32:27-kcryptd/253:0]
root	974764	2 0 14:18 ?	00:00:00 [kworker/u32:28-kcryptd/253:0]
root	974765	2 0 14:18 ?	00:00:00 [kworker/u32:29-events_unbound]
root	974766	2 0 14:18 ?	00:00:00 [kworker/u32:30+events_unbound]
root	974767	2 0 14:18 ?	00:00:00 [kworker/u32:31-events_unbound]
root	974768	2 0 14:18 ?	00:00:00 [kworker/u32:32-events_unbound]
root	974769	2 0 14:18 ?	00:00:00 [kworker/u32:33-events_unbound]
root	974770	2 0 14:18 ?	00:00:00 [kworker/u32:34-events_unbound]
root	974771	2 0 14:18 ?	00:00:00 [kworker/u32:35-events_unbound]
root	974772	2 0 14:18 ?	00:00:00 [kworker/u32:36-events_unbound]
root	974773	2 0 14:18 ?	00:00:00 [kworker/u32:37-events_unbound]
root	974774	2 0 14:18 ?	00:00:00 [kworker/u32:38-events_unbound]
root	974775	2 0 14:18 ?	00:00:00 [kworker/u32:39-events_unbound]
root	974776	2 0 14:18 ?	00:00:00 [kworker/u32:40-kcryptd/253:0]
root	974778	2 0 14:18 ?	00:00:00 [kworker/u32:42-events_unbound]
root	974779	2 0 14:18 ?	00:00:00 [kworker/3:3-events]
root	974780	2 0 14:18 ?	00:00:00 [kworker/3:4-events]
root	974798	2 0 14:18 ?	00:00:00 [kworker/3:5-events]
root	974995	2 0 14:18 ?	00:00:00 [kworker/u33:2-rb_allocator]
root	975505	2 0 14:20 ?	00:00:00 [kworker/6:3-events]
root	975656	2 0 14:20 ?	00:00:00 [kworker/5:3-events]
root	975657	2 0 14:29 ?	00:00:00 [kworker/1:3-pm]
root	975658	2 0 14:29 ?	00:00:00 [kworker/1:4-events]

```

abhijit 975722 351306 0 14:29 ? 00:00:02 /usr/lib/firefox/firefox -contentproc -childID 1836 -isForBrowser -
prefsLen 35227 -prefMapSize 247458 -jsInitLen 229864 -parentBuildID 20240108143603 -greomni /usr/lib/firefox/omni.ja -
appomni /usr/lib/firefox/browser/omni.ja -appDir /usr/lib/firefox/browser {d9ea12ea-5b3f-477c-9c08-9a4196ea8d04} 351306
true tab
root 976242 2 0 15:16 ? 00:00:00 [kworker/0:0-events]
root 976243 2 0 15:16 ? 00:00:00 [kworker/0:3-events]
root 976244 2 0 15:16 ? 00:00:00 [kworker/0:4-events]
postfix 976248 3508 0 15:16 ? 00:00:00 pickup -l -t unix -u -c
abhijit 976770 351306 0 15:18 ? 00:00:00 /usr/lib/firefox/firefox -contentproc -childID 1840 -isForBrowser -
prefsLen 35227 -prefMapSize 247458 -jsInitLen 229864 -parentBuildID 20240108143603 -greomni /usr/lib/firefox/omni.ja -
appomni /usr/lib/firefox/browser/omni.ja -appDir /usr/lib/firefox/browser {cb567d37-1c03-46db-966c-632f4b708354} 351306
true tab
abhijit 976836 351306 0 15:19 ? 00:00:00 /usr/lib/firefox/firefox -contentproc -childID 1841 -isForBrowser -
prefsLen 35227 -prefMapSize 247458 -jsInitLen 229864 -parentBuildID 20240108143603 -greomni /usr/lib/firefox/omni.ja -
appomni /usr/lib/firefox/browser/omni.ja -appDir /usr/lib/firefox/browser {b62e1986-221e-481e-bda4-62fb37caa67} 351306
true tab
abhijit 977138 351306 0 15:20 ? 00:00:00 /usr/lib/firefox/firefox -contentproc -childID 1842 -isForBrowser -
prefsLen 35227 -prefMapSize 247458 -jsInitLen 229864 -parentBuildID 20240108143603 -greomni /usr/lib/firefox/omni.ja -
appomni /usr/lib/firefox/browser/omni.ja -appDir /usr/lib/firefox/browser {989da731-7bd5-44ce-abd0-200ca4c84b9b} 351306
true tab
abhijit 977184 594726 0 15:21 pts/5 00:00:00 ps -eaf
abhijit 1664880 3629 0 Jan07 ? 00:00:06 /usr/bin/gnome-calendar --gapplication-service
abhijit 1665340 3629 0 Jan07 ? 00:00:00 /usr/bin/gpg-agent --supervised
abhijit 3872409 3629 0 Jan07 ? 00:00:05 /usr/bin/seahorse --gapplication-service
root 3873244 1 0 Jan07 ? 00:00:55 /sbin/mount.ntfs /dev/nvme0n1p3 /media/abhijit/windows -o
rw,nodev,nosuid,windows_names,uid=1000,gid=1000,uhelper=udisks2
abhijit 3884359 30565 0 Jan07 pts/7 00:00:00 bash
abhijit 4108623 30565 0 Jan08 pts/8 00:00:00 bash
abhijit 4136834 3890 0 Jan08 ? 00:00:00 /usr/lib/libreoffice/program/oosplash --calc
abhijit 4136869 4136834 0 Jan08 ? 00:32:11 /usr/lib/libreoffice/program/soffice.bin --calc
abhijit 4139495 30565 0 Jan08 pts/9 00:00:00 bash

```

The PID of the grand-parent of the process with PID 4108623 is :

3629



The two processes which were created by kernel have PIDs (in increasing order)

0

and

1



The process that created most of the "graphical" processes is having PID

2



Question 6

Partially correct

Mark 0.67 out of 1.00

Order the following events in boot process (from 1 onwards)

Shell	5	✗
Boot loader	2	✓
Init	4	✓
BIOS	1	✓
Login interface	6	✗
OS	3	✓

Your answer is partially correct.

You have correctly selected 4.

The correct answer is: Shell → 6, Boot loader → 2, Init → 4, BIOS → 1, Login interface → 5, OS → 3

Question 7

Correct

Mark 0.50 out of 0.50

Is the terminal a part of the kernel on GNU/Linux systems?

- a. yes
 b. no ✓ wrong

The correct answer is: no

Question 8

Correct

Mark 1.00 out of 1.00

Consider the following programs

exec1.c

```
#include <unistd.h>
#include <stdio.h>
int main() {
    execl("./exec2", "./exec2", NULL);
}
```

exec2.c

```
#include <unistd.h>
#include <stdio.h>
int main() {
    execl("/bin/ls", "/bin/ls", NULL);
    printf("hello\n");
}
```

Compiled as

```
cc  exec1.c -o exec1
cc  exec2.c -o exec2
```

And run as

```
$ ./exec1
```

Explain the output of the above command (./exec1)

Assume that /bin/ls , i.e. the 'ls' program exists.

Select one:

- a. Execution fails as the call to execl() in exec1 fails
- b. Execution fails as the call to execl() in exec2 fails
- c. Execution fails as one exec can't invoke another exec
- d. Program prints hello
- e. "ls" runs on current directory✓

Your answer is correct.

The correct answer is: "ls" runs on current directory

Question 9

Correct

Mark 0.50 out of 0.50

When you turn your computer ON, on BIOS based systems, you are often shown an option like "Press F9 for boot options". What does this mean?

- a. The choice of booting slowly or fast
- b. The BIOS allows us to choose the boot device, the device from which the boot loader will be loaded✓
- c. The choice of which OS to boot from
- d. The choice of the boot loader (e.g. GRUB or Windows-Loader)

The correct answer is: The BIOS allows us to choose the boot device, the device from which the boot loader will be loaded

Question 10

Correct

Mark 1.00 out of 1.00

How does the distinction between kernel mode and user mode function as a rudimentary form of protection (security) ?

Select one:

- a. It prohibits invocation of kernel code completely, if a user program is running
- b. It prohibits one process from accessing other process's memory
- c. It disallows hardware interrupts when a process is running
- d. It prohibits a user mode process from running privileged instructions✓

Your answer is correct.

The correct answer is: It prohibits a user mode process from running privileged instructions

Question 11

Correct

Mark 1.00 out of 1.00

Select all the correct statements about bootloader.

Every wrong selection will deduct marks proportional to $1/n$ where n is total wrong choices in the question.

You will get minimum a zero.

- a. Bootloader must be one sector in length
- b. The bootloader loads the BIOS
- c. Bootloaders allow selection of OS to boot from✓
- d. Modern Bootloaders often allow configuring the way an OS boots✓
- e. LILO is a bootloader✓

Your answer is correct.

The correct answers are: LILO is a bootloader, Modern Bootloaders often allow configuring the way an OS boots, Bootloaders allow selection of OS to boot from

Question 12

Correct

Mark 1.00 out of 1.00

Predict the output of the program given here.

Assume that all the path names for the programs are correct. For example "/usr/bin/echo" will actually run echo command.

Assume that there is no mixing of printf output on screen if two of them run concurrently.

In the answer replace a new line by a single space.

For example::

good
output
should be written as good output

--

```
main() {  
    int i;  
    i = fork();  
    if(i == 0)  
        execl("/usr/bin/echo", "/usr/bin/echo", "hi", 0);  
    else  
        wait(0);  
    fork();  
    execl("/usr/bin/echo", "/usr/bin/echo", "one", 0);  
}
```

Answer: hi one one ✓

The correct answer is: hi one one

Question 13

Correct

Mark 1.00 out of 1.00

Select all statements that correctly explain the use/purpose of system calls.

Select one or more:

- a. Provide an environment for process creation ✓
- b. Handle exceptions like division by zero
- c. Allow I/O device access to user processes ✓
- d. Provide services for accessing files ✓
- e. Handle ALL types of interrupts
- f. Run each instruction of an application program
- g. Switch from user mode to kernel mode ✓

Your answer is correct.

The correct answers are: Switch from user mode to kernel mode, Provide services for accessing files, Allow I/O device access to user processes, Provide an environment for process creation

Question 14

Partially correct

Mark 0.60 out of 1.00

Select all the correct statements about two modes of CPU operation

Select one or more:

- a. Some instructions are allowed to run only in user mode, while all instructions can run in kernel mode
- b. There is an instruction like 'iret' to return from kernel mode to user mode✓
- c. The two modes are essential for a multiprogramming system✓
- d. The two modes are essential for a multitasking system✓
- e. The software interrupt instructions change the mode from user mode to kernel mode and jumps to predefined location simultaneously

Your answer is partially correct.

You have correctly selected 3.

The correct answers are: The two modes are essential for a multiprogramming system, The two modes are essential for a multitasking system, There is an instruction like 'iret' to return from kernel mode to user mode, The software interrupt instructions change the mode from user mode to kernel mode and jumps to predefined location simultaneously, Some instructions are allowed to run only in user mode, while all instructions can run in kernel mode

Question 15

Partially correct

Mark 0.25 out of 0.50

Select all the correct statements about bootloader.

Every wrong selection will deduct marks proportional to $1/n$ where n is total wrong choices in the question.

You will get minimum a zero.

- a. Bootloaders allow selection of OS to boot from✓
- b. Bootloader must be one sector in length✗
- c. LILO is a bootloader✓
- d. The bootloader loads the BIOS
- e. Modern Bootloaders often allow configuring the way an OS boots✓

Your answer is partially correct.

You have selected too many options.

The correct answers are: LILO is a bootloader, Modern Bootloaders often allow configuring the way an OS boots, Bootloaders allow selection of OS to boot from

Question 16

Partially correct

Mark 0.33 out of 1.00

Select the sequence of events that are NOT possible, assuming a non-interruptible kernel code

(Note: non-interruptible kernel code means, if the kernel code is executing, then interrupts will be disabled).

Note: A possible sequence may have some missing steps in between. An impossible sequence will have n and n+1th steps such that n+1th step can not follow n'th step.

Select one or more:

- a. P1 running

P1 makes system call and blocks

Scheduler

P2 running

P2 makes system call and blocks

Scheduler

P3 running

Hardware interrupt

Interrupt unblocks P1

Interrupt returns

P3 running

Timer interrupt

Scheduler

P1 running

- b. P1 running

Keyboard hardware interrupt

Keyboard interrupt handler running

interrupt handler returns

P1 running

P1 makes system call

System call returns

P1 running

timer interrupt

Scheduler

P2 running

- c. P1 running ✓

P1 makes system call and blocks

Scheduler

P2 running

P2 makes system call and blocks

Scheduler

P1 running again

- d.

P1 running

P1 makes system call

Scheduler

P2 running

P2 makes system call and blocks

Scheduler

P1 running again

- e. P1 running

P1 makes system call

System call returns

P1 running

timer interrupt

Scheduler running

P2 running

- f. P1 running

P1 makes system call

timer interrupt

Scheduler

P2 running

timer interrupt

Scheuler

P1 running

P1's system call return

Your answer is partially correct.

You have correctly selected 1.

The correct answers are: P1 running

P1 makes system call and blocks

Scheduler

P2 running

P2 makes system call and blocks

Scheduler

P1 running again, P1 running

P1 makes system call

timer interrupt

Scheduler

P2 running

timer interrupt

Scheduler

P1 running

P1's system call return,

P1 running

P1 makes system call

Scheduler

P2 running

P2 makes system call and blocks

Scheduler

P1 running again

Question 17

Correct

Mark 1.00 out of 1.00

What will this program do?

```
int main() {  
    fork();  
    execl("/bin/ls", "/bin/ls", NULL);  
    printf("hello");  
}
```

- a. run ls twice and print hello twice
- b. run ls twice and print hello twice, but output will appear in some random order
- c. run ls twice✓
- d. one process will run ls, another will print hello
- e. run ls once

Your answer is correct.

The correct answer is: run ls twice

[◀ Surprise Quiz - 1 \(pre-requisites\)](#)

Jump to...

[Surprise Quiz - 3 \(processes, memory management, event driven kernel\), compilation-linking-loading](#) ▶

Started on Wednesday, 7 February 2024, 6:09 PM

State Finished

Completed on Wednesday, 7 February 2024, 7:10 PM

Time taken 1 hour

Grade 18.14 out of 20.00 (90.68%)

Question 1

Correct

Mark 1.00 out of 1.00

Consider the following code and MAP the file to which each fd points at the end of the code. Assume that files/folders exist when needed with proper permissions and open() calls work.

```
int main(int argc, char *argv[]) {
    int fd1, fd2 = 1, fd3 = 1, fd4 = 1;

    fd1 = open("/tmp/1", O_WRONLY | O_CREAT, S_IRUSR|S_IWUSR);
    fd2 = open("/tmp/2", O_RDONLY);
    fd3 = open("/tmp/3", O_WRONLY | O_CREAT, S_IRUSR|S_IWUSR);
    close(0);
    close(1);
    dup(fd2);
    dup(fd3);
    close(fd3);
    dup2(fd2, fd4);
    printf("%d %d %d %d\n", fd1, fd2, fd3, fd4);
    return 0;
}
```

fd2	/tmp/2	✓
fd1	/tmp/1	✓
0	/tmp/2	✓
fd3	closed	✓
2	stderr	✓
1	/tmp/3	✓
fd4	/tmp/2	✓

The correct answer is: fd2 → /tmp/2, fd1 → /tmp/1, 0 → /tmp/2, fd3 → closed, 2 → stderr, 1 → /tmp/3, fd4 → /tmp/2

Question 2

Partially correct

Mark 1.43 out of 2.00

Order the events that occur on a timer interrupt:

Save the context of the currently running process	3	✓
Jump to scheduler code	4	✓
Set the context of the new process	6	✓
Jump to a code pointed by IDT	1	✗
Change to kernel stack of currently running process	2	✗
Select another process for execution	5	✓
Execute the code of the new process	7	✓

The correct answer is: Save the context of the currently running process → 3, Jump to scheduler code → 4, Set the context of the new process → 6, Jump to a code pointed by IDT → 2, Change to kernel stack of currently running process → 1, Select another process for execution → 5, Execute the code of the new process → 7

Question 3

Partially correct

Mark 0.75 out of 1.00

Select the sequence of events that are NOT possible, assuming an interruptible kernel code

Select one or more:

- a. P1 running

P1 makes system call and blocks

Scheduler

P2 running

P2 makes system call and blocks

Scheduler

P3 running

Hardware interrupt

Interrupt unblocks P1

Interrupt returns

P3 running

Timer interrupt

Scheduler

P1 running

- b. P1 running

keyboard hardware interrupt

keyboard interrupt handler running

interrupt handler returns

P1 running

P1 makes system call

system call returns

P1 running

timer interrupt

scheduler

P2 running

- c. P1 running ✖

P1 makes system call

timer interrupt

Scheduler

P2 running

timer interrupt

Scheuler

P1 running

P1's system call return

- d. P1 running ✓

P1 makes system call and blocks

Scheduler

P2 running

P2 makes system call and blocks

Scheduler

P1 running again

- e. ✓

P1 running

P1 makes system call

Scheduler

P2 running

P2 makes system call and blocks

Scheduler

P1 running again

- f. P1 running

P1 makes system call

system call returns

P1 running
timer interrupt
Scheduler running
P2 running

The correct answers are: P1 running

P1 makes system call and blocks

Scheduler

P2 running

P2 makes system call and blocks

Scheduler

P1 running again,

P1 running

P1 makes system call

Scheduler

P2 running

P2 makes system call and blocks

Scheduler

P1 running again

Question 4

Partially correct

Mark 1.60 out of 2.00

Match the elements of C program to their place in memory

Allocated Memory	Heap	✓
Global variables	Data	✓
Arguments	Stack	✓
#include files	No Memory needed	✗
Local Static variables	Data	✓
Global Static variables	Data	✓
Function code	Code	✓
Local Variables	Stack	✓
#define MACROS	No Memory needed	✓
Code of main()	Main_Code	✗

The correct answer is: Allocated Memory → Heap, Global variables → Data, Arguments → Stack, #include files → No memory needed, Local Static variables → Data, Global Static variables → Data, Function code → Code, Local Variables → Stack, #define MACROS → No Memory needed, Code of main() → Code

Question 5

Correct

Mark 1.00 out of 1.00

Select the order in which the various stages of a compiler execute.

Intermediate code generation	3	✓
Syntactical Analysis	2	✓
Pre-processing	1	✓
Linking	4	✓
Loading	does not exist	✓

The correct answer is: Intermediate code generation → 3, Syntactical Analysis → 2, Pre-processing → 1, Linking → 4, Loading → does not exist

Question 6

Correct

Mark 2.00 out of 2.00

Consider the two programs given below to implement the command (ignore the fact that error checks are not done on return values of functions)

```
$ ls . /tmp/asdfksdf >/tmp/ddd 2>&1
```

Program 1

```
int main(int argc, char *argv[]) {  
    int fd, n, i;  
    char buf[128];  
  
    fd = open("/tmp/ddd", O_WRONLY | O_CREAT, S_IRUSR | S_IWUSR);  
    close(1);  
    dup(fd);  
    close(2);  
    dup(fd);  
    execl("/bin/ls", "/bin/ls", ".", "/tmp/asldjfaldfs", NULL);  
}
```

Program 2

```
int main(int argc, char *argv[]) {  
    int fd, n, i;  
    char buf[128];  
  
    close(1);  
    fd = open("/tmp/ddd", O_WRONLY | O_CREAT, S_IRUSR | S_IWUSR);  
    close(2);  
    fd = open("/tmp/ddd", O_WRONLY | O_CREAT, S_IRUSR | S_IWUSR);  
    execl("/bin/ls", "/bin/ls", ".", "/tmp/asldjfaldfs", NULL);  
}
```

Select all the correct statements about the programs

Select one or more:

- a. Program 1 ensures 2>&1 and does not ensure >/tmp/ddd
- b. Program 1 is correct for > /tmp/ddd but not for 2>&1
- c. Only Program 1 is correct✓
- d. Program 1 does 1>&2
- e. Program 2 does 1>&2
- f. Both program 1 and 2 are incorrect
- g. Program 2 makes sure that there is one file offset used for '2' and '1'
- h. Program 2 is correct for > /tmp/ddd but not for 2>&1
- i. Program 1 makes sure that there is one file offset used for '2' and '1'✓
- j. Only Program 2 is correct
- k. Program 2 ensures 2>&1 and does not ensure > /tmp/ddd
- l. Both programs are correct

The correct answers are: Only Program 1 is correct, Program 1 makes sure that there is one file offset used for '2' and '1'

Question 7

Correct

Mark 1.00 out of 1.00

Select all the correct statements about zombie processes

Select one or more:

- a. A process can become zombie if it finishes, but the parent has finished before it✓
- b. init() typically keeps calling wait() for zombie processes to get cleaned up✓
- c. A process becomes zombie when it finishes, and remains zombie until parent calls wait() on it✓
- d. Zombie processes are harmless even if OS is up for long time
- e. A zombie process remains zombie forever, as there is no way to clean it up
- f. If the parent of a process finishes, before the process itself, then after finishing the process is typically attached to 'init' as parent✓
- g. A process becomes zombie when its parent finishes
- h. A zombie process occupies space in OS data structures✓

The correct answers are: A process becomes zombie when it finishes, and remains zombie until parent calls wait() on it, A process can become zombie if it finishes, but the parent has finished before it, A zombie process occupies space in OS data structures, If the parent of a process finishes, before the process itself, then after finishing the process is typically attached to 'init' as parent, init() typically keeps calling wait() for zombie processes to get cleaned up

Question 8

Correct

Mark 1.00 out of 1.00

Consider the image given below, which explains how paging works.

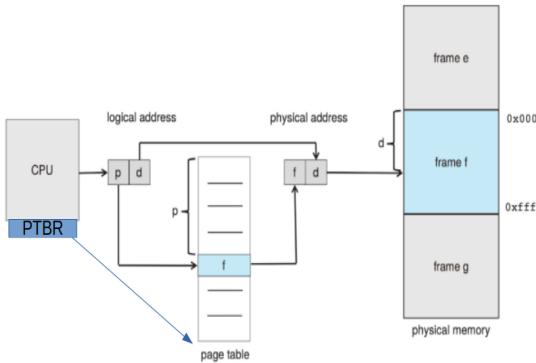


Figure 9.8 Paging hardware.

Mention whether each statement is True or False, with respect to this image.

True	False	
<input checked="" type="radio"/>	<input type="radio"/>	The page table is itself present in Physical memory
<input checked="" type="radio"/>	<input type="radio"/>	Maximum Size of page table is determined by number of bits used for page number
<input checked="" type="radio"/>	<input type="radio"/>	The page table is indexed using page number
<input type="radio"/>	<input checked="" type="radio"/>	The page table is indexed using frame number
<input checked="" type="radio"/>	<input type="radio"/>	The PTBR is present in the CPU as a register
<input checked="" type="radio"/>	<input type="radio"/>	The physical address may not be of the same size (in bits) as the logical address
<input type="radio"/>	<input checked="" type="radio"/>	Size of page table is always determined by the size of RAM
<input type="radio"/>	<input checked="" type="radio"/>	The locating of the page table using PTBR also involves paging translation

The page table is itself present in Physical memory: True

Maximum Size of page table is determined by number of bits used for page number: True

The page table is indexed using page number: True

The page table is indexed using frame number: False

The PTBR is present in the CPU as a register: True

The physical address may not be of the same size (in bits) as the logical address: True

Size of page table is always determined by the size of RAM: False

The locating of the page table using PTBR also involves paging translation: False

Question 9

Correct

Mark 1.00 out of 1.00

Select all the correct statements about MMU and its functionality (on a non-demand paged system)

Select one or more:

- a. The operating system interacts with MMU for every single address translation
- b. Illegal memory access is detected by operating system
- c. Logical to physical address translations in MMU are done in hardware, automatically ✓
- d. MMU is a separate chip outside the processor
- e. MMU is inside the processor ✓
- f. Illegal memory access is detected in hardware by MMU and a trap is raised ✓
- g. The Operating system sets up relevant CPU registers to enable proper MMU translations ✓
- h. Logical to physical address translations in MMU are done with specific machine instructions

The correct answers are: MMU is inside the processor, Logical to physical address translations in MMU are done in hardware, automatically, The Operating system sets up relevant CPU registers to enable proper MMU translations, Illegal memory access is detected in hardware by MMU and a trap is raised

Question 10

Partially correct

Mark 0.86 out of 1.00

Mark the statements as True/False w.r.t. the basic concepts of memory management.

True	False	
<input type="radio"/> ✗	<input checked="" type="radio"/> ✗	When a process is executing, each virtual address is converted into physical address by the kernel directly.
<input checked="" type="radio"/> ✗	<input type="radio"/> ✗	The compiler generates address references for code/data/stack/heap in the executable file, depending on the MM architecture provided by CPU and kernel.
<input type="radio"/> ✗	<input checked="" type="radio"/> ✗	The compiler generates the address references for code/data/stack/heap in the executable file as per the memory management schema chosen by the compiler itself, and then the kernel ensures that program is executed with this schema.
<input type="radio"/> ✗	<input checked="" type="radio"/> ✗	The compiler interacts with the kernel continuously while compiling a program and obtains the correct set of memory addresses for code/stack/heap/data and then generates the machine code file.
<input checked="" type="radio"/> ✗	<input type="radio"/> ✗	The kernel ensures that the MMU is setup before scheduling a process and then the CPU/MMU ensures that the address translation takes place.
<input checked="" type="radio"/> ✗	<input type="radio"/> ✗	When a process is executing, each virtual address is converted into physical address by the CPU hardware directly.
<input type="radio"/> ✗	<input checked="" type="radio"/> ✗	The kernel refers to the page table for converting each virtual address to physical address.

When a process is executing, each virtual address is converted into physical address by the kernel directly.: False

The compiler generates address references for code/data/stack/heap in the executable file, depending on the MM architecture provided by CPU and kernel.: True

The compiler generates the address references for code/data/stack/heap in the executable file as per the memory management schema chosen by the compiler itself, and then the kernel ensures that program is executed with this schema.: False

The compiler interacts with the kernel continuously while compiling a program and obtains the correct set of memory addresses for code/stack/heap/data and then generates the machine code file.: False

The kernel ensures that the MMU is setup before scheduling a process and then the CPU/MMU ensures that the address translation takes place.: True

When a process is executing, each virtual address is converted into physical address by the CPU hardware directly.: True

The kernel refers to the page table for converting each virtual address to physical address.: False

Question 11

Partially correct

Mark 0.50 out of 1.00

Select the correct statements about paging (not demand paging) mechanism

Select one or more:

- a. User process can update its own PTBR
- b. An invalid entry on a page means, it was an illegal memory reference
- c. The PTBR is loaded by the OS ✓
- d. Page table is accessed by the OS as part of execution of an instruction
- e. User process can update its own page table entries
- f. Page table is accessed by the MMU as part of execution of an instruction ✓
- g. OS creates the page table for every process ✓
- h. An invalid entry on a page means, either it was illegal memory reference or the page was not present in memory. ✗

The correct answers are: OS creates the page table for every process, The PTBR is loaded by the OS, Page table is accessed by the MMU as part of execution of an instruction, An invalid entry on a page means, it was an illegal memory reference

Question 12

Correct

Mark 1.00 out of 1.00

Select the compiler's view of the process's address space, for each of the following MMU schemes:

(Assume that each scheme, e.g. paging/segmentation/etc is effectively utilised)

Segmentation	many continuous chunks of variable size	✓
Segmentation, then paging	many continuous chunks of variable size	✓
Relocation + Limit	one continuous chunk	✓
Paging	one continuous chunk	✓

The correct answer is: Segmentation → many continuous chunks of variable size, Segmentation, then paging → many continuous chunks of variable size, Relocation + Limit → one continuous chunk, Paging → one continuous chunk

Question 13

Correct

Mark 1.00 out of 1.00

Select the state that is not possible after the given state, for a process:

New:	Running	✓
Ready :	Waiting	✓
Running:	None of these	✓
Waiting:	Running	✓

Question 14

Correct

Mark 1.00 out of 1.00

A process blocks itself means

- a. The kernel code of system call, called by the process, moves the process to a waiting queue and calls scheduler✓
- b. The application code calls the scheduler
- c. The kernel code of an interrupt handler, moves the process to a waiting queue and calls scheduler
- d. The kernel code of system call calls scheduler

The correct answer is: The kernel code of system call, called by the process, moves the process to a waiting queue and calls scheduler

Question 15

Correct

Mark 1.00 out of 1.00

which of the following is not a difference between real mode and protected mode

- a. in real mode the segment is multiplied by 16, in protected mode segment is used as index in GDT
- b. in real mode the addressable memory is less than in protected mode
- c. in real mode general purpose registers are 16 bit, in protected mode they are 32 bit
- d. processor starts in real mode
- e. in real mode the addressable memory is more than in protected mode✓

The correct answer is: in real mode the addressable memory is more than in protected mode

Question 16

Correct

Mark 1.00 out of 1.00

Predict the output of the program given here.

Assume that there is no mixing of printf output on screen if two of them run concurrently.

In the answer replace a new line by a single space.

For example::

good
output
should be written as good output

--

```
int main() {  
    int pid;  
    printf("hi\n");  
    pid = fork();  
    if(pid == 0) {  
        exit(0);  
    }  
    printf("bye\n");  
    fork();  
    printf("ok\n");  
}
```

Answer: hi bye ok ok



The correct answer is: hi bye ok ok

Question 17

Correct

Mark 1.00 out of 1.00

Which of the following are NOT a part of job of a typical compiler?

- a. Invoke the linker to link the function calls with their code, extern globals with their declaration
- b. Check the program for syntactical errors
- c. Convert high level language code to machine code
- d. Suggest alternative pieces of code that can be written ✓
- e. Check the program for logical errors ✓
- f. Process the # directives in a C program

The correct answers are: Check the program for logical errors, Suggest alternative pieces of code that can be written

[◀ Surprise Quiz - 2](#)

Jump to...

[Questions for test on kalloc/kfree/kvmalloc, etc. ►](#)

Started on Saturday, 20 February 2021, 2:51 PM

State Finished

Completed on Saturday, 20 February 2021, 3:55 PM

Time taken 1 hour 3 mins

Grade 7.30 out of 20.00 (37%)

Question 1

Partially correct

Mark 0.80 out of 1.00

Select all the correct statements about the state of a process.

- a. A process can self-terminate only when it's running ✓
- b. Typically, it's represented as a number in the PCB ✓
- c. A process that is running is not on the ready queue ✓
- d. Processes in the ready queue are in the ready state ✓
- e. It is not maintained in the data structures by kernel, it is only for conceptual understanding of programmers
- f. Changing from running state to waiting state results in "giving up the CPU" ✓
- g. A process in ready state is ready to receive interrupts
- h. A waiting process starts running after the wait is over ✗
- i. A process changes from running to ready state on a timer interrupt ✓
- j. A process in ready state is ready to be scheduled ✓
- k. A running process may terminate, or go to wait or become ready again ✓
- l. A process waiting for I/O completion is typically woken up by the particular interrupt handler code ✓
- m. A process waiting for any condition is woken up by another process only
- n. A process changes from running to ready state on a timer interrupt or any I/O wait

Your answer is partially correct.

You have selected too many options.

The correct answers are: Typically, it's represented as a number in the PCB, A process in ready state is ready to be scheduled, Processes in the ready queue are in the ready state, A process that is running is not on the ready queue, A running process may terminate, or go to wait or become ready again, A process changes from running to ready state on a timer interrupt, Changing from running state to waiting state results in "giving up the CPU", A process can self-terminate only when it's running, A process waiting for I/O completion is typically woken up by the particular interrupt handler code

Question 2

Incorrect

Mark 0.00 out of 1.00

For each line of code mentioned on the left side, select the location of sp/esp that is in use

`jmp *%eax`
in entry.S

0x7c00 to 0x10000



`ljmp $(SEG_KCODE<<3), $start32`
in bootasm.S

0x10000 to 0x7c00



`call bootmain`
in bootasm.S

0x7c00 to 0x10000



`cli`
in bootasm.S

0x7c00 to 0



`readseg((uchar*)elf, 4096, 0);`
in bootmain.c

The 4KB area in kernel image, loaded in memory, named as 'stack'



Your answer is incorrect.

The correct answer is: `jmp *%eax`

`in entry.S` → The 4KB area in kernel image, loaded in memory, named as 'stack', `ljmp $(SEG_KCODE<<3), $start32`

`in bootasm.S` → Immaterail as the stack is not used here, `call bootmain`

`in bootasm.S` → 0x7c00 to 0, `cli`

`in bootasm.S` → Immaterail as the stack is not used here, `readseg((uchar*)elf, 4096, 0);`

`in bootmain.c` → 0x7c00 to 0

Question 3

Correct

Mark 0.25 out of 0.25

Order the following events in boot process (from 1 onwards)

Boot loader	2	✓
Shell	6	✓
BIOS	1	✓
OS	3	✓
Init	4	✓
Login interface	5	✓

Your answer is correct.

The correct answer is: Boot loader → 2, Shell → 6, BIOS → 1, OS → 3, Init → 4, Login interface → 5

Question 4

Partially correct

Mark 0.30 out of 0.50

Consider the following command and its output:

```
$ ls -lht xv6.img kernel
-rw-rw-r-- 1 abhijit abhijit 4.9M Feb 15 11:09 xv6.img
-rwxrwxr-x 1 abhijit abhijit 209K Feb 15 11:09 kernel*
```

Following code in bootmain()

```
readseg((uchar*)elf, 4096, 0);
```

and following selected lines from Makefile

```
xv6.img: bootblock kernel
dd if=/dev/zero of=xv6.img count=10000
dd if=bootblock of=xv6.img conv=notrunc
dd if=kernel of=xv6.img seek=1 conv=notrunc
```

```
kernel: $(OBJS) entry.o entryother initcode kernel.ld
$(LD) $(LDFLAGS) -T kernel.ld -o kernel entry.o $(OBJS) -b binary initcode entryother
$(OBJDUMP) -S kernel > kernel.asm
$(OBJDUMP) -t kernel | sed '1,/SYMBOL TABLE/d; s/ .* / /; /^$$/d' > kernel.sym
```

Also read the code of bootmain() in xv6 kernel.

Select the options that describe the meaning of these lines and their correlation.

- a. Although the size of the kernel file is 209 Kb, only 4Kb out of it is the actual kernel code and remaining part is all zeroes.
- b. The kernel is compiled by linking multiple .o files created from .c files; and the entry.o, initcode, entryother files ✓
- c. The kernel.ld file contains instructions to the linker to link the kernel properly ✓
- d. The bootmain() code does not read the kernel completely in memory
- e. readseg() reads first 4k bytes of kernel in memory
- f. Although the size of the xv6.img file is ~5MB, only some part out of it is the bootloader+kernel code and remaining part is all zeroes.
- g. The kernel.asm file is the final kernel file
- h. The kernel disk image is ~5MB, the kernel within it is 209 kb, but bootmain() initially reads only first 4kb, and the later part is not read as it is user programs.
- i. The kernel disk image is ~5MB, the kernel within it is 209 kb, but bootmain() initially reads only first 4kb, and the later part is read using program headers in bootmain(). ✓

Your answer is partially correct.

You have correctly selected 3.

The correct answers are: The kernel disk image is ~5MB, the kernel within it is 209 kb, but bootmain() initially reads only first 4kb, and the later part is read using program headers in bootmain(), readseg() reads first 4k bytes of kernel in memory, The kernel is compiled by linking multiple .o files created from .c files; and the entry.o, initcode, entryother files, The kernel.ld file contains instructions to the linker to link the kernel properly, Although the size of the xv6.img file is ~5MB, only some part out of it is the bootloader+kernel code and remaining part is all zeroes.

Question 5

Partially correct

Mark 0.50 out of 1.00

```
int f() {  
    int count;  
    for (count = 0; count < 2; count++) {  
        if (fork() == 0)  
            printf("Operating-System\n");  
    }  
    printf("TYCOMP\n");  
}
```

The number of times "Operating-System" is printed, is:

Answer:

The correct answer is: 7.00

Question 6

Partially correct

Mark 0.40 out of 0.50

Select Yes/True if the mentioned element must be a part of PCB

Select No/False otherwise.

Yes	No	
<input checked="" type="radio"/>	<input type="radio"/> X	PID
<input checked="" type="radio"/>	<input type="radio"/> X	Process context
<input checked="" type="radio"/>	<input type="radio"/> X	List of opened files
<input checked="" type="radio"/>	<input type="radio"/> X	Process state
<input type="radio"/> X	<input checked="" type="radio"/>	Parent's PID
<input type="radio"/> X	<input checked="" type="radio"/>	Pointer to IDT
<input type="radio"/> X	<input checked="" type="radio"/>	Function pointers to all system calls
<input checked="" type="radio"/>	<input type="radio"/> X	Memory management information about that process
<input checked="" type="radio"/>	<input checked="" type="radio"/>	Pointer to the parent process
<input checked="" type="radio"/>	<input type="radio"/> X	EIP at the time of context switch

PID: Yes

Process context: Yes

List of opened files: Yes

Process state: Yes

Parent's PID: No

Pointer to IDT: No

Function pointers to all system calls: No

Memory management information about that process: Yes

Pointer to the parent process: Yes

EIP at the time of context switch: Yes

Question 7

Incorrect

Mark 0.00 out of 1.00

Select all the correct statements about code of bootmain() in xv6

```

void
bootmain(void)
{
    struct elfhdr *elf;
    struct proghdr *ph, *eph;
    void (*entry)(void);
    uchar* pa;

    elf = (struct elfhdr*)0x10000; // scratch space

    // Read 1st page off disk
    readseg((uchar*)elf, 4096, 0);

    // Is this an ELF executable?
    if(elf->magic != ELF_MAGIC)
        return; // let bootasm.S handle error

    // Load each program segment (ignores ph flags).
    ph = (struct proghdr*)((uchar*)elf + elf->phoff);
    eph = ph + elf->phnum;
    for(; ph < eph; ph++){
        pa = (uchar*)ph->paddr;
        readseg(pa, ph->filesz, ph->off);
        if(ph->memsz > ph->filesz)
            stosb(pa + ph->filesz, 0, ph->memsz - ph->filesz);
    }

    // Call the entry point from the ELF header.
    // Does not return!
    entry = (void(*)(void))(elf->entry);
    entry();
}

```

Also, inspect the relevant parts of the xv6 code. binary files, etc and run commands as you deem fit to answer this question.

- a. The kernel file gets loaded at the Physical address 0x10000 +0x80000000 in memory. ✗
- b. The elf->entry is set by the linker in the kernel file and it's 0x80000000 ✗
- c. The kernel ELF file contains actual physical address where particular sections of 'kernel' file should be loaded ✓
- d. The kernel file in memory is not necessarily a continuously filled in chunk, it may have holes in it. ✓
- e. The kernel file has only two program headers ✓
- f. The elf->entry is set by the linker in the kernel file and it's 0x80000000 ✗
- g. The readseg finally invokes the disk I/O code using assembly instructions ✓
- h. The elf->entry is set by the linker in the kernel file and it's 8010000c ✓
- i. The kernel file gets loaded at the Physical address 0x10000 in memory. ✓
- j. The condition if(ph->memsz > ph->filesz) is never true. ✗
- k. The stosb() is used here, to fill in some space in memory with zeroes ✓

Your answer is incorrect.

The correct answers are: The kernel file gets loaded at the Physical address 0x10000 in memory., The kernel file in memory is not necessarily a continuously filled in chunk, it may have holes in it., The elf->entry is set by the linker in the kernel file and it's 8010000c, The readseg finally invokes the disk I/O code using assembly instructions, The stosb() is used here, to fill in some space in memory with zeroes, The kernel ELF file contains actual physical address where particular sections of 'kernel' file should be loaded, The kernel file has only two program headers

Question 8

Partially correct

Mark 0.13 out of 0.25

Which of the following are NOT a part of job of a typical compiler?

- a. Check the program for logical errors ✓
- b. Convert high level language code to machine code
- c. Process the # directives in a C program
- d. Invoke the linker to link the function calls with their code, extern globals with their declaration
- e. Check the program for syntactical errors
- f. Suggest alternative pieces of code that can be written

Your answer is partially correct.

You have correctly selected 1.

The correct answers are: Check the program for logical errors, Suggest alternative pieces of code that can be written

Question 9

Correct

Mark 0.25 out of 0.25

Rank the following storage systems from slowest (first) to fastest(last)

Cache	6	✓
Hard Disk	3	✓
RAM	5	✓
Optical Disks	2	✓
Non volatile memory	4	✓
Registers	7	✓
Magnetic Tapes	1	✓

Your answer is correct.

The correct answer is: Cache → 6, Hard Disk → 3, RAM → 5, Optical Disks → 2, Non volatile memory → 4, Registers → 7, Magnetic Tapes → 1

Question 10

Partially correct

Mark 0.21 out of 0.50

Which of the following parts of a C program do not have any corresponding machine code ?

- a. local variable declaration
- b. global variables
- c. function calls ✗
- d. #directives ✓
- e. expressions
- f. pointer dereference
- g. typedefs ✓

Your answer is partially correct.

You have correctly selected 2.

The correct answers are: #directives, typedefs, global variables

Question 11

Correct

Mark 0.25 out of 0.25

Match a system call with it's description

pipe	create an unnamed FIFO storage with 2 ends - one for reading and another for writing	✓
dup	create a copy of the specified file descriptor into smallest available file descriptor	✓
dup2	create a copy of the specified file descriptor into another specified file descriptor	✓
exec	execute a binary file overlaying the image of current process	✓
fork	create an identical child process	✓

Your answer is correct.

The correct answer is: pipe → create an unnamed FIFO storage with 2 ends - one for reading and another for writing, dup → create a copy of the specified file descriptor into smallest available file descriptor, dup2 → create a copy of the specified file descriptor into another specified file descriptor, exec → execute a binary file overlaying the image of current process, fork → create an identical child process

Question 12

Correct

Mark 0.25 out of 0.25

Match the register with the segment used with it.

eip	cs	✓
edi	es	✓
esi	ds	✓
ebp	ss	✓
esp	ss	✓

Your answer is correct.

The correct answer is: eip → cs, edi → es, esi → ds, ebp → ss, esp → ss

Question 13

Correct

Mark 0.25 out of 0.25

What's the trapframe in xv6?

- a. A frame of memory that contains all the trap handler code
- b. The sequence of values, including saved registers, constructed on the stack when an interrupt occurs, built by hardware only
- c. The IDT table
- d. A frame of memory that contains all the trap handler code's function pointers
- e. A frame of memory that contains all the trap handler's addresses
- f. The sequence of values, including saved registers, constructed on the stack when an interrupt occurs, built by hardware + code in trapasm.S ✓
- g. The sequence of values, including saved registers, constructed on the stack when an interrupt occurs, built by code in trapasm.S only

Your answer is correct.

The correct answer is: The sequence of values, including saved registers, constructed on the stack when an interrupt occurs, built by hardware + code in trapasm.S

Question 14

Incorrect

Mark 0.00 out of 0.50

Select all the correct statements about linking and loading.

Select one or more:

- a. Continuous memory management schemes can support dynamic linking and dynamic loading. ✗
- b. Loader is last stage of the linker program ✗
- c. Continuous memory management schemes can support static linking and dynamic loading. (may be inefficiently) ✓
- d. Dynamic linking and loading is not possible without demand paging or demand segmentation. ✓
- e. Dynamic linking essentially results in relocatable code. ✓
- f. Continuous memory management schemes can support static linking and static loading. (may be inefficiently) ✓
- g. Loader is part of the operating system ✓
- h. Static linking leads to non-relocatable code ✗
- i. Dynamic linking is possible with continuous memory management, but variable sized partitions only. ✗

Your answer is incorrect.

The correct answers are: Continuous memory management schemes can support static linking and static loading. (may be inefficiently), Continuous memory management schemes can support static linking and dynamic loading. (may be inefficiently), Dynamic linking essentially results in relocatable code., Loader is part of the operating system, Dynamic linking and loading is not possible without demand paging or demand segmentation.

Question 15

Incorrect

Mark 0.00 out of 0.25

In bootasm.S, on the line

```
ljmp    $(SEG_KCODE<<3), $start32
```

The SEG_KCODE << 3, that is shifting of 1 by 3 bits is done because

- a. The value 8 is stored in code segment
- b. The code segment is 16 bit and only upper 13 bits are used for segment number
- c. The code segment is 16 bit and only lower 13 bits are used for segment number ✗
- d. While indexing the GDT using CS, the value in CS is always divided by 8
- e. The ljmp instruction does a divide by 8 on the first argument

Your answer is incorrect.

The correct answer is: The code segment is 16 bit and only upper 13 bits are used for segment number

Question 16

Partially correct

Mark 0.07 out of 0.50

Order the events that occur on a timer interrupt:

Change to kernel stack

1	✗
---	---

Jump to a code pointed by IDT

2	✗
---	---

Jump to scheduler code

5	✗
---	---

Set the context of the new process

4	✗
---	---

Save the context of the currently running process

3	✓
---	---

Execute the code of the new process

6	✗
---	---

Select another process for execution

7	✗
---	---

Your answer is partially correct.

You have correctly selected 1.

The correct answer is: Change to kernel stack → 2, Jump to a code pointed by IDT → 1, Jump to scheduler code → 4, Set the context of the new process → 6, Save the context of the currently running process → 3, Execute the code of the new process → 7, Select another process for execution → 5

Question 17

Incorrect

Mark 0.00 out of 1.00

Consider the two programs given below to implement the command (ignore the fact that error checks are not done on return values of functions)

```
$ ls . /tmp/asdfksdf >/tmp/ddd 2>&1
```

Program 1

```
int main(int argc, char *argv[]) {
    int fd, n, i;
    char buf[128];

    fd = open("/tmp/ddd", O_WRONLY | O_CREAT, S_IRUSR | S_IWUSR);
    close(1);
    dup(fd);
    close(2);
    dup(fd);
    execl("/bin/ls", "/bin/ls", ".", "/tmp/asldjfaldfs", NULL);
}
```

Program 2

```
int main(int argc, char *argv[]) {
    int fd, n, i;
    char buf[128];

    close(1);
    fd = open("/tmp/ddd", O_WRONLY | O_CREAT, S_IRUSR | S_IWUSR);
    close(2);
    fd = open("/tmp/ddd", O_WRONLY | O_CREAT, S_IRUSR | S_IWUSR);
    execl("/bin/ls", "/bin/ls", ".", "/tmp/asldjfaldfs", NULL);
}
```

Select all the correct statements about the programs

Select one or more:

- a. Both programs are correct ✗
- b. Program 2 makes sure that there is one file offset used for '2' and '1' ✗
- c. Only Program 2 is correct ✗
- d. Program 2 does 1>&2 ✗
- e. Program 2 ensures 2>&1 and does not ensure >/tmp/ddd ✗
- f. Program 1 makes sure that there is one file offset used for '2' and '1' ✓
- g. Program 1 is correct for >/tmp/ddd but not for 2>&1 ✗
- h. Program 1 does 1>&2 ✗
- i. Both program 1 and 2 are incorrect ✗
- j. Program 2 is correct for >/tmp/ddd but not for 2>&1 ✗
- k. Only Program 1 is correct ✓
- l. Program 1 ensures 2>&1 and does not ensure >/tmp/ddd ✗

Your answer is incorrect.

The correct answers are: Only Program 1 is correct, Program 1 makes sure that there is one file offset used for '2' and '1'

Question 18

Correct

Mark 0.25 out of 0.25

Select the option which best describes what the CPU does during its powered ON lifetime

- a. Ask the user what is to be done, and execute that task
- b. Ask the OS what is to be done, and execute that task
- c. Fetch instructions specified by location given by PC, Decode and Execute it, during execution increment PC or change PC as per the instruction itself, Ask the User or the OS what is to be done next, repeat
- d. Fetch instructions specified by location given by PC, Decode and Execute it, during execution increment PC or change PC as per ✓ the instruction itself, repeat
- e. Fetch instruction specified by OS, Decode and execute it, repeat
- f. Fetch instructions specified by location given by PC, Decode and Execute it, during execution increment PC or change PC as per the instruction itself, Ask OS what is to be done next, repeat

The correct answer is: Fetch instructions specified by location given by PC, Decode and Execute it, during execution increment PC or change PC as per the instruction itself, repeat

Question 19

Partially correct

Mark 0.86 out of 1.00

Consider the following code and MAP the file to which each fd points at the end of the code.

```
int main(int argc, char *argv[]) {
    int fd1, fd2 = 1, fd3 = 1, fd4 = 1;

    fd1 = open("/tmp/1", O_WRONLY | O_CREAT, S_IRUSR|S_IWUSR);
    fd2 = open("/tmp/2", O_RDONLY);
    fd3 = open("/tmp/3", O_WRONLY | O_CREAT, S_IRUSR|S_IWUSR);
    close(0);
    close(1);
    dup(fd2);
    dup(fd3);
    close(fd3);
    dup2(fd2, fd4);
    printf("%d %d %d %d\n", fd1, fd2, fd3, fd4);
    return 0;
}
```

1	closed	✗
fd4	/tmp/2	✓
fd2	/tmp/2	✓
fd1	/tmp/1	✓
2	stderr	✓
0	/tmp/2	✓
fd3	closed	✓

Your answer is partially correct.

You have correctly selected 6.

The correct answer is: 1 → /tmp/3, fd4 → /tmp/2, fd2 → /tmp/2, fd1 → /tmp/1, 2 → stderr, 0 → /tmp/2, fd3 → closed

Question 20

Incorrect

Mark 0.00 out of 2.00

Following code claims to implement the command

/bin/ls -l | /usr/bin/head -3 | /usr/bin/tail -1

Fill in the blanks to make the code work.

Note: Do not include space in writing any option. x[1][2] should be written without any space, and so is the case with [1] or [2]. Pay attention to exact syntax and do not write any extra character like ';' or = etc.

```
int main(int argc, char *argv[]) {
```

```
    int pid1, pid2;
```

```
    int pfd[
```

```
    1
```

```
    x ] [2];
```

```
    pipe(
```

```
    2
```

```
    x );
```

```
    pid1 =
```

```
    3
```

```
    x ;
```

```
    if(pid1 != 0) {
```

```
        close(pfd[0]
```

```
        0
```

```
    x );
```

```
    close(
```

```
    pid1
```

```
    x );
```

```
    dup(
```

```
    pid2
```

```
    x );
```

```
    execl("/bin/ls", "/bin/ls", "
```

```
    1
```

```
    x ", NULL);
```

```
    }
```

```
    pipe(
```

```
    
```

```
    x );
```

```
    
```

```
    x = fork();
```

```
    if(pid2 == 0) {
```

```
        close(
```

```
        
```

```
    x ;
```

```
        close(0);
```

```
        dup(
```

```
        
```

```
    x );
```

```
        close(pfd[1]
```

```
        
```

```
✗ );
close(
  
✗ );
dup(
  
✗ );
execl("/usr/bin/head", "/usr/bin/head", "  
  
✗ ", NULL);
} else {
close(pfd
  
✗ );
close(
  
✗ );
dup(
  
✗ );
close(pfd
  
✗ );
execl("/usr/bin/tail", "/usr/bin/tail", "  
  
✗ ", NULL);
}  
}
```

Question 21

Partially correct

Mark 0.11 out of 1.00

Select all the correct statements about calling convention on x86 32-bit.

- a. Return address is one location above the ebp ✓
- b. Parameters may be passed in registers or on stack ✓
- c. Space for local variables is allocated by subtracting the stack pointer inside the code of the called function ✓
- d. The ebp pointers saved on the stack constitute a chain of activation records ✓
- e. The two lines in the beginning of each function, "push %ebp; mov %esp, %ebp", create space for local variables ✗
- f. Parameters may be passed in registers or on stack ✓
- g. The return value is either stored on the stack or returned in the eax register ✗
- h. Parameters are pushed on the stack in left-right order
- i. during execution of a function, ebp is pointing to the old ebp
- j. Space for local variables is allocated by subtracting the stack pointer inside the code of the caller function ✗
- k. Compiler may allocate more memory on stack than needed ✓

Your answer is partially correct.

You have selected too many options.

The correct answers are: Compiler may allocate more memory on stack than needed, Parameters may be passed in registers or on stack, Return address is one location above the ebp, during execution of a function, ebp is pointing to the old ebp, Space for local variables is allocated by subtracting the stack pointer inside the code of the called function, The ebp pointers saved on the stack constitute a chain of activation records

Question 22

Correct

Mark 1.00 out of 1.00

Match the program with its output (ignore newlines in the output. Just focus on the count of the number of 'hi')

main() { int i = fork(); if(i == 0) execl("/usr/bin/echo", "/usr/bin/echo", "hi\n", NULL); }

hi ✓

main() { fork(); execl("/usr/bin/echo", "/usr/bin/echo", "hi\n", NULL); }

hi hi ✓

main() { int i = NULL; fork(); printf("hi\n"); }

hi hi ✓

main() { execl("/usr/bin/echo", "/usr/bin/echo", "hi\n", NULL); }

hi ✓

Your answer is correct.

The correct answer is: main() { int i = fork(); if(i == 0) execl("/usr/bin/echo", "/usr/bin/echo", "hi\n", NULL); } → hi, main() { fork(); execl("/usr/bin/echo", "/usr/bin/echo", "hi\n", NULL); } → hi hi, main() { int i = NULL; fork(); printf("hi\n"); } → hi hi, main() { execl("/usr/bin/echo", "/usr/bin/echo", "hi\n", NULL); } → hi

Question 23

Incorrect

Mark 0.00 out of 0.50

Some part of the bootloader of xv6 is written in assembly while some part is written in C. Why is that so?

Select all the appropriate choices

- a. The code in assembly is required for transition to protected mode, from real mode; but calling convention was applicable all the time ✗
- b. The setting up of the most essential memory management infrastructure needs assembly code ✓
- c. The code for reading ELF file can not be written in assembly ✗
- d. The code in assembly is required for transition to protected mode, from real mode; after that calling convention applies, hence code can be written in C ✓

Your answer is incorrect.

The correct answers are: The code in assembly is required for transition to protected mode, from real mode; after that calling convention applies, hence code can be written in C, The setting up of the most essential memory management infrastructure needs assembly code

Question 24

Incorrect

Mark 0.00 out of 0.50

xv6.img: bootblock kernel

```
dd if=/dev/zero of=xv6.img count=10000
dd if=bootblock of=xv6.img conv=notrunc
dd if=kernel of=xv6.img seek=1 conv=notrunc
```

Consider above lines from the Makefile. Which of the following is incorrect?

- a. The size of the kernel file is nearly 5 MB ✓
- b. The kernel is located at block-1 of the xv6.img ✗
- c. The xv6.img is of the size 10,000 blocks of 512 bytes each and occupies 10,000 blocks on the disk. ✗
- d. The size of xv6.img is exactly = (size of bootblock) + (size of kernel) ✗
- e. The bootblock is located on block-0 of the xv6.img ✗
- f. The xv6.img is of the size 10,000 blocks of 512 bytes each and occupies upto 10,000 blocks on the disk. ✓
- g. The bootblock may be 512 bytes or less (looking at the Makefile instruction) ✗
- h. The xv6.img is the virtual disk that is created by combining the bootblock and the kernel file. ✗
- i. The size of the xv6.img is nearly 5 MB ✗
- j. xv6.img is the virtual processor used by the qemu emulator ✓
- k. Blocks in xv6.img after kernel may be all zeroes. ✗

Your answer is incorrect.

The correct answers are: xv6.img is the virtual processor used by the qemu emulator, The xv6.img is of the size 10,000 blocks of 512 bytes each and occupies upto 10,000 blocks on the disk., The size of the kernel file is nearly 5 MB, The size of xv6.img is exactly = (size of bootblock) + (size of kernel)

Question 25

Incorrect

Mark 0.00 out of 1.00

Select the sequence of events that are NOT possible, assuming a non-interruptible kernel code

Select one or more:

a. P1 running

P1 makes system call

timer interrupt

Scheduler

P2 running

timer interrupt

Scheuler

P1 running

P1's system call return

b. P1 running

P1 makes sytem call and blocks

Scheduler

P2 running

P2 makes sytem call and blocks

Scheduler

P1 running again



c. P1 running

P1 makes system call

system call returns

P1 running

timer interrupt

Scheduler running

P2 running

d. P1 running

P1 makes sytem call and blocks

Scheduler

P2 running

P2 makes sytem call and blocks

Scheduler

P3 running

Hardware interrupt

Interrupt unblocks P1

Interrupt returns

P3 running

Timer interrupt

Scheduler

P1 running



e.

P1 running

P1 makes sytem call

Scheduler

P2 running

P2 makes sytem call and blocks

Scheduler

P1 running again

f. P1 running

keyboard hardware interrupt

keyboard interrupt handler running

interrupt handler returns

P1 running

P1 makes sytem call

system call returns



P1 running
timer interrupt
scheduler
P2 running

Your answer is incorrect.

The correct answers are: P1 running

P1 makes system call and blocks

Scheduler

P2 running

P2 makes system call and blocks

Scheduler

P1 running again, P1 running

P1 makes system call

timer interrupt

Scheduler

P2 running

timer interrupt

Scheuler

P1 running

P1's system call return,

P1 running

P1 makes system call

Scheduler

P2 running

P2 makes system call and blocks

Scheduler

P1 running again

Question 26

Correct

Mark 0.25 out of 0.25

Which of the following are the files related to bootloader in xv6?

- a. bootasm.s and entry.S
- b. bootasm.S and bootmain.c ✓
- c. bootasm.S, bootmain.c and bootblock.c
- d. bootmain.c and bootblock.S

Your answer is correct.

The correct answer is: bootasm.S and bootmain.c

Question 27

Correct

Mark 0.25 out of 0.25

Match the following parts of a C program to the layout of the process in memory

Instructions	Text section	✓
Local Variables	Stack Section	✓
Dynamically allocated memory	Heap Section	✓
Global and static data	Data section	✓

Your answer is correct.

The correct answer is:

Instructions → Text section, Local Variables → Stack Section,
Dynamically allocated memory → Heap Section,
Global and static data → Data section

Question 28

Incorrect

Mark 0.00 out of 0.50

What will this program do?

```
int main() {  
    fork();  
    execl("/bin/ls", "/bin/ls", NULL);  
    printf("hello");  
}
```

- a. one process will run ls, another will print hello
- b. run ls once ✗
- c. run ls twice
- d. run ls twice and print hello twice
- e. run ls twice and print hello twice, but output will appear in some random order

Your answer is incorrect.

The correct answer is: run ls twice

Question 29

Correct

Mark 0.25 out of 0.25

What is the OS Kernel?

- a. The code that controls hardware, abstracts access to hardware resources using system calls, creates an environment for processes to be created and run ✓ correct
- b. The set of tools like compiler, linker, loader, terminal, shell, etc.
- c. Only the system programs like compiler, linker, loader, etc.
- d. Everything that I see on my screen

The correct answer is: The code that controls hardware, abstracts access to hardware resources using system calls, creates an environment for processes to be created and run

Question 30

Correct

Mark 0.50 out of 0.50

Which of the following is/are not saved during context switch?

- a. Program Counter
- b. General Purpose Registers
- c. Bus ✓
- d. Stack Pointer
- e. MMU related registers/information
- f. Cache ✓
- g. TLB ✓

Your answer is correct.

The correct answers are: TLB, Cache, Bus

Question 31

Partially correct

Mark 0.10 out of 0.25

Select the order in which the various stages of a compiler execute.

Linking	3	
Syntactical Analysis	2	
Pre-processing	1	
Intermediate code generation	does not exist	
Loading	4	

Your answer is partially correct.

You have correctly selected 2.

The correct answer is: Linking → 4, Syntactical Analysis → 2, Pre-processing → 1, Intermediate code generation → 3, Loading → does not exist

Question 32

Partially correct

Mark 0.08 out of 0.50

Order the sequence of events, in scheduling process P1 after process P0

context of P0 is saved in P0's PCB	2	
context of P1 is loaded from P1's PCB	3	
Process P1 is running	5	
timer interrupt occurs	6	
Process P0 is running	1	
Control is passed to P1	4	

Your answer is partially correct.

You have correctly selected 1.

The correct answer is: context of P0 is saved in P0's PCB → 3, context of P1 is loaded from P1's PCB → 4, Process P1 is running → 6, timer interrupt occurs → 2, Process P0 is running → 1, Control is passed to P1 → 5

Question 33

Not answered

Marked out of 1.00

Select the correct statements about interrupt handling in xv6 code

- a. On any interrupt/syscall/exception the control first jumps in vectors.S
- b. The trapframe pointer in struct proc, points to a location on user stack
- c. Each entry in IDT essentially gives the values of CS and EIP to be used in handling that interrupt
- d. xv6 uses the 64th entry in IDT for system calls
- e. The CS and EIP are changed only after pushing user code's SS,ESP on stack
- f. The trapframe pointer in struct proc, points to a location on kernel stack
- g. The function trap() is called only in case of hardware interrupt
- h. The CS and EIP are changed only immediately on a hardware interrupt
- i. All the 256 entries in the IDT are filled

- j. On any interrupt/syscall/exception the control first jumps in trapasm.S
- k. The function trap() is called irrespective of hardware interrupt/system-call/exception
- l. xv6 uses the 0x64th entry in IDT for system calls
- m. Before going to alltraps, the kernel stack contains upto 5 entries.

Your answer is incorrect.

The correct answers are: All the 256 entries in the IDT are filled, Each entry in IDT essentially gives the values of CS and EIP to be used in handling that interrupt, xv6 uses the 64th entry in IDT for system calls, On any interrupt/syscall/exception the control first jumps in trapasm.S, Before going to alltraps, the kernel stack contains upto 5 entries., The trapframe pointer in struct proc, points to a location on kernel stack, The function trap() is called irrespective of hardware interrupt/system-call/exception, The CS and EIP are changed only after pushing user code's SS,ESP on stack

[◀ \(Assignment\) Change free list management in xv6](#)

Jump to...

Started on Thursday, 18 March 2021, 2:46 PM

State Finished

Completed on Thursday, 18 March 2021, 3:50 PM

Time taken 1 hour 4 mins

Grade 10.36 out of 20.00 (52%)

Question 1

Partially correct

Mark 0.57 out of 1.00

Mark True, the actions done as part of code of swtch() in swtch.S, in xv6

True

False

<input checked="" type="radio"/>	<input checked="" type="radio"/>	Restore new callee saved registers from kernel stack of new context	✓
<input checked="" type="radio"/>	<input checked="" type="radio"/>	Save old callee saved registers on kernel stack of old context	✓
<input checked="" type="radio"/>	<input checked="" type="radio"/>	Save old callee saved registers on user stack of old context	✓
<input checked="" type="radio"/>	<input checked="" type="radio"/>	Switch from old process context to new process context	✗
<input checked="" type="radio"/>	<input checked="" type="radio"/>	Switch from one stack (old) to another(new)	✗
<input checked="" type="radio"/>	<input checked="" type="radio"/>	Restore new callee saved registers from user stack of new context	✓
<input checked="" type="radio"/>	<input checked="" type="radio"/>	Jump to code in new context	✗

Restore new callee saved registers from kernel stack of new context: True

Save old callee saved registers on kernel stack of old context: True

Save old callee saved registers on user stack of old context: False

Switch from old process context to new process context: False

Switch from one stack (old) to another(new): True

Restore new callee saved registers from user stack of new context: False

Jump to code in new context: False

Question 2

Partially correct

Mark 0.17 out of 0.50

For each function/code-point, select the status of segmentation setup in xv6

bootmain()	gdt setup with 3 entries, right from first line of code of bootloader	✗
kvmalloc() in main()	gdt setup with 5 entries (0 to 4) on one processor	✗
after startothers() in main()	gdt setup with 5 entries (0 to 4) on all processors	✓
after seginit() in main()	gdt setup with 5 entries (0 to 4) on all processors	✗
bootasm.S	gdt setup with 3 entries, right from first line of code of bootloader	✗
entry.S	gdt setup with 3 entries, at start32 symbol of bootasm.S	✓

Your answer is partially correct.

You have correctly selected 2.

The correct answer is: bootmain() → gdt setup with 3 entries, at start32 symbol of bootasm.S, kvmalloc() in main() → gdt setup with 3 entries, at start32 symbol of bootasm.S, after startothers() in main() → gdt setup with 5 entries (0 to 4) on all processors, after seginit() in main() → gdt setup with 5 entries (0 to 4) on one processor, bootasm.S → gdt setup with 3 entries, at start32 symbol of bootasm.S, entry.S → gdt setup with 3 entries, at start32 symbol of bootasm.S

Question 3

Partially correct

Mark 0.38 out of 1.00

Compare paging with demand paging and select the correct statements.

Select one or more:

- a. The meaning of valid-invalid bit in page table is different in paging and demand-paging. ✓
- b. Demand paging requires additional hardware support, compared to paging. ✓
- c. Paging requires some hardware support in CPU
- d. With paging, it's possible to have user programs bigger than physical memory. ✗
- e. Both demand paging and paging support shared memory pages. ✓
- f. Demand paging always increases effective memory access time.
- g. With demand paging, it's possible to have user programs bigger than physical memory. ✓
- h. Calculations of number of bits for page number and offset are same in paging and demand paging. ✓
- i. TLB hit ration has zero impact in effective memory access time in demand paging.
- j. Paging requires NO hardware support in CPU

Your answer is partially correct.

You have correctly selected 5.

The correct answers are: Demand paging requires additional hardware support, compared to paging., Both demand paging and paging support shared memory pages., With demand paging, it's possible to have user programs bigger than physical memory., Demand paging always increases effective memory access time., Paging requires some hardware support in CPU, Calculations of number of bits for page number and offset are same in paging and demand paging., The meaning of valid-invalid bit in page table is different in paging and demand-paging.

Question 4

Partially correct

Mark 0.44 out of 0.50

Suppose a processor supports base(relocation register) + limit scheme of MMU.

Assuming this, mark the statements as True/False

True	False	
<input checked="" type="radio"/>	<input type="radio"/> <input checked="" type="checkbox"/>	The OS may terminate the process while handling the interrupt of memory violation
<input checked="" type="radio"/>	<input type="radio"/> <input checked="" type="checkbox"/>	The hardware detects any memory access beyond the limit value and raises an interrupt
<input type="radio"/> <input checked="" type="checkbox"/>	<input type="radio"/>	The hardware may terminate the process while handling the interrupt of memory violation
<input checked="" type="radio"/>	<input type="radio"/> <input checked="" type="checkbox"/>	The OS sets up the relocation and limit registers when the process is scheduled
<input checked="" type="radio"/>	<input type="radio"/> <input checked="" type="checkbox"/>	The compiler generates machine code assuming continuous memory address space for process, and calculating appropriate sizes for code, and data;
<input type="radio"/> <input checked="" type="checkbox"/>	<input type="radio"/> <input checked="" type="checkbox"/>	The process sets up its own relocation and limit registers when the process is scheduled
<input type="radio"/> <input checked="" type="checkbox"/>	<input type="radio"/> <input checked="" type="checkbox"/>	The OS detects any memory access beyond the limit value and raises an interrupt
<input type="radio"/> <input checked="" type="checkbox"/>	<input type="radio"/> <input checked="" type="checkbox"/>	The compiler generates machine code assuming appropriately sized segments for code, data and stack.

The OS may terminate the process while handling the interrupt of memory violation: True

The hardware detects any memory access beyond the limit value and raises an interrupt: True

The hardware may terminate the process while handling the interrupt of memory violation: False

The OS sets up the relocation and limit registers when the process is scheduled: True

The compiler generates machine code assuming continuous memory address space for process, and calculating appropriate sizes for code, and data;: True

The process sets up its own relocation and limit registers when the process is scheduled: False

The OS detects any memory access beyond the limit value and raises an interrupt: False

The compiler generates machine code assuming appropriately sized segments for code, data and stack.: False

Question 5

Correct

Mark 0.50 out of 0.50

Consider the following list of free chunks, in continuous memory management:

10k, 25k, 12k, 7k, 9k, 13k

Suppose there is a request for chunk of size 9k, then the free chunk selected under each of the following schemes will be

Best fit:

9k



First fit:

10k



Worst fit:

25k

**Question 6**

Partially correct

Mark 0.50 out of 1.00

Select all the correct statements about MMU and its functionality

Select one or more:

- a. MMU is a separate chip outside the processor
- b. MMU is inside the processor ✓
- c. Logical to physical address translations in MMU are done with specific machine instructions
- d. The operating system interacts with MMU for every single address translation ✗
- e. Illegal memory access is detected in hardware by MMU and a trap is raised ✓
- f. The Operating system sets up relevant CPU registers to enable proper MMU translations
- g. Logical to physical address translations in MMU are done in hardware, automatically ✓
- h. Illegal memory access is detected by operating system

Your answer is partially correct.

You have correctly selected 3.

The correct answers are: MMU is inside the processor, Logical to physical address translations in MMU are done in hardware, automatically, The Operating system sets up relevant CPU registers to enable proper MMU translations, Illegal memory access is detected in hardware by MMU and a trap is raised

Question 7

Incorrect

Mark 0.00 out of 0.50

Assuming a 8- KB page size, what is the page numbers for the address 874815 reference in decimal :
(give answer also in decimal)

Answer: ✖

The correct answer is: 107

Question 8

Incorrect

Mark 0.00 out of 0.25

Select the compiler's view of the process's address space, for each of the following MMU schemes:
(Assume that each scheme,e.g. paging/segmentation/etc is effectively utilised)

Segmentation, then paging	<input type="checkbox"/> Many continuous chunks each of page size	✖
Relocation + Limit	<input type="checkbox"/> Many continuous chunks of same size	✖
Segmentation	<input type="checkbox"/> one continuous chunk	✖
Paging	<input type="checkbox"/> many continuous chunks of variable size	✖

Your answer is incorrect.

The correct answer is: Segmentation, then paging → many continuous chunks of variable size, Relocation + Limit → one continuous chunk, Segmentation → many continuous chunks of variable size, Paging → one continuous chunk

Question 9

Incorrect

Mark 0.00 out of 0.50

Suppose the memory access time is 180ns and TLB hit ratio is 0.3, then effective memory access time is (in nanoseconds);

Answer: ✖

The correct answer is: 306.00

Question 10

Correct

Mark 0.50 out of 0.50

In xv6, The struct context is given as

```
struct context {
    uint edi;
    uint esi;
    uint ebx;
    uint ebp;
    uint eip;
};
```

Select all the reasons that explain why only these 5 registers are included in the struct context.

- a. The segment registers are same across all contexts, hence they need not be saved ✓
- b. esp is not saved in context, because context{} is on stack and it's address is always argument to swtch() ✓
- c. xv6 tries to minimize the size of context to save memory space
- d. esp is not saved in context, because it's not part of the context
- e. eax, ecx, edx are caller save, hence no need to save ✓

Your answer is correct.

The correct answers are: The segment registers are same across all contexts, hence they need not be saved, eax, ecx, edx are caller save, hence no need to save, esp is not saved in context, because context{} is on stack and it's address is always argument to swtch()

Question 11

Partially correct

Mark 0.83 out of 1.50

Arrange the following events in order, in page fault handling:

Disk interrupt wakes up the process

7	✓
---	---

The reference bit is found to be invalid by MMU

1	✓
---	---

OS makes available an empty frame

6	✗
---	---

Restart the instruction that caused the page fault

9	✓
---	---

A hardware interrupt is issued

3	✗
---	---

OS schedules a disk read for the page (from backing store)

5	✓
---	---

Process is kept in wait state

4	✗
---	---

Page tables are updated for the process

8	✓
---	---

Operating system decides that the page was not in memory

2	✗
---	---

Your answer is partially correct.

You have correctly selected 5.

The correct answer is: Disk interrupt wakes up the process → 7, The reference bit is found to be invalid by MMU → 1, OS makes available an empty frame → 4, Restart the instruction that caused the page fault → 9, A hardware interrupt is issued → 2, OS schedules a disk read for the page (from backing store) → 5, Process is kept in wait state → 6, Page tables are updated for the process → 8, Operating system decides that the page was not in memory → 3

Question 12

Incorrect

Mark 0.00 out of 0.50

Suppose a kernel uses a buddy allocator. The smallest chunk that can be allocated is of size 32 bytes. One bit is used to track each such chunk, where 1 means allocated and 0 means free. The chunk looks like this as of now:

00001010

Now, there is a request for a chunk of 70 bytes.

After this allocation, the bitmap, indicating the status of the buddy allocator will be

Answer: 11101010



The correct answer is: 11111010

Question 13

Incorrect

Mark 0.00 out of 0.25

The complete range of virtual addresses (after main() in main.c is over), from which the free pages used by kalloc() and kfree() is derived, are:

- a. end, 4MB
- b. P2V(end), P2V(PHYSTOP)
- c. end, P2V(4MB + PHYSTOP)
- d. P2V(end), PHYSTOP ✗
- e. end, (4MB + PHYSTOP)
- f. end, PHYSTOP
- g. end, P2V(PHYSTOP)

Your answer is incorrect.

The correct answer is: end, P2V(PHYSTOP)

Question 14

Partially correct

Mark 0.33 out of 0.50

Match the pair

Hashed page table	Linear search on collision done by OS (e.g. SPARC Solaris) typically	✓
Inverted Page table	Linear/Parallel search using frame number in page table	✗
Hierarchical Paging	More memory access time per hierarchy	✓

Your answer is partially correct.

You have correctly selected 2.

The correct answer is: Hashed page table → Linear search on collision done by OS (e.g. SPARC Solaris) typically, Inverted Page table → Linear/Parallel search using page number in page table, Hierarchical Paging → More memory access time per hierarchy

Question 15

Partially correct

Mark 0.29 out of 0.50

After virtual memory is implemented

(select T/F for each of the following) One Program's size can be larger than physical memory size

True	False	
<input checked="" type="radio"/>	<input type="radio"/> ✗	Code need not be completely in memory
<input checked="" type="radio"/>	<input type="radio"/> ✗	Cumulative size of all programs can be larger than physical memory size
<input type="radio"/> ✗	<input checked="" type="radio"/>	Virtual access to memory is granted
<input checked="" type="radio"/>	<input type="radio"/> ✗	Logical address space could be larger than physical address space
<input type="radio"/> ✗	<input checked="" type="radio"/>	Virtual addresses are available
<input checked="" type="radio"/>	<input checked="" type="radio"/> ✗	Relatively less I/O may be possible during process execution
<input checked="" type="radio"/>	<input type="radio"/> ✗	One Program's size can be larger than physical memory size

Code need not be completely in memory: True

Cumulative size of all programs can be larger than physical memory size: True

Virtual access to memory is granted: False

Logical address space could be larger than physical address space: True

Virtual addresses are available: False

Relatively less I/O may be possible during process execution: True

One Program's size can be larger than physical memory size: True

Question 16

Partially correct

Mark 0.64 out of 1.00

W.r.t. Memory management in xv6,

xv6 uses physical memory upto 224 MB only
Mark statements True or False**True****False**

<input checked="" type="radio"/>	<input type="radio"/> <input checked="" type="checkbox"/>	The switchkvm() call in scheduler() is invoked after control comes to it from sched(), thus demanding execution in kernel's context	✓
<input checked="" type="radio"/>	<input type="radio"/> <input checked="" type="checkbox"/>	The stack allocated in entry.S is used as stack for scheduler's context for first processor	✓
<input checked="" type="radio"/>	<input type="radio"/> <input checked="" type="checkbox"/>	The switchkvm() call in scheduler() changes CR3 to use page directory kpgdir	✓
<input checked="" type="radio"/>	<input type="radio"/> <input checked="" type="checkbox"/>	The free page-frame are created out of nearly 222 MB	✗
<input checked="" type="radio"/>	<input type="radio"/> <input checked="" type="checkbox"/>	The kernel code and data take up less than 2 MB space	✓
<input checked="" type="radio"/>	<input type="radio"/> <input checked="" type="checkbox"/>	The switchkvm() call in scheduler() changes CR3 to use page directory of new process	✗
<input checked="" type="radio"/>	<input type="radio"/> <input checked="" type="checkbox"/>	The switchkvm() call in scheduler() is invoked after control comes to it from swtch() scheduler(), thus demanding execution in new process's context	✓
<input checked="" type="radio"/>	<input type="radio"/> <input checked="" type="checkbox"/>	PHYSTOP can be increased to some extent, simply by editing memlayout.h	✓
<input checked="" type="radio"/>	<input type="radio"/> <input checked="" type="checkbox"/>	xv6 uses physical memory upto 224 MB only	✗
<input checked="" type="radio"/>	<input type="radio"/> <input checked="" type="checkbox"/>	The process's address space gets mapped on frames, obtained from ~2MB:224MB range	✓
<input checked="" type="radio"/>	<input type="radio"/> <input checked="" type="checkbox"/>	The kernel's page table given by kpgdir variable is used as stack for scheduler's context	✗

The switchkvm() call in scheduler() is invoked after control comes to it from sched(), thus demanding execution in kernel's context: True

The stack allocated in entry.S is used as stack for scheduler's context for first processor: True

The switchkvm() call in scheduler() changes CR3 to use page directory kpgdir: True

The free page-frame are created out of nearly 222 MB: True

The kernel code and data take up less than 2 MB space: True

The switchkvm() call in scheduler() changes CR3 to use page directory of new process: False

The switchkvm() call in scheduler() is invoked after control comes to it from swtch() scheduler(), thus demanding execution in new process's context: False

PHYSTOP can be increased to some extent, simply by editing memlayout.h: True

xv6 uses physical memory upto 224 MB only: True

The process's address space gets mapped on frames, obtained from ~2MB:224MB range: True

The kernel's page table given by kpgdir variable is used as stack for scheduler's context: False

Question 17

Incorrect

Mark 0.00 out of 1.50

Consider the reference string

6 4 2 0 1 2 6 9 2 0 5

If the number of page frames is 3, then total number of page faults (including initial), using LRU replacement is:

Answer: ✖

#6# 6,4# 6,4,2 # 0,4,2#0,1,2#6,1,2#6,9,2#0,9,2#0,5,2

The correct answer is: 9

Question 18

Partially correct

Mark 0.31 out of 0.50

Consider the image given below, which explains how paging works.

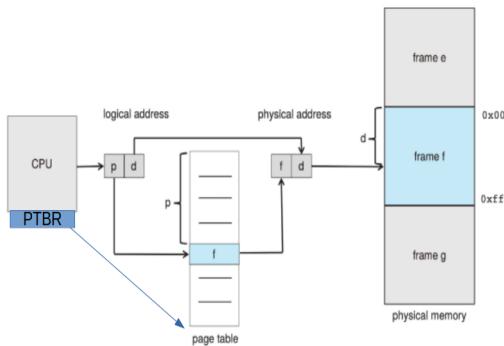


Figure 9.8 Paging hardware.

Mention whether each statement is True or False, with respect to this image.

True	False	
<input checked="" type="radio"/>	<input type="radio"/>	The PTBR is present in the CPU as a register
<input type="radio"/>	<input checked="" type="radio"/>	The page table is indexed using frame number
<input checked="" type="radio"/>	<input type="radio"/>	The page table is indexed using page number
<input type="radio"/>	<input checked="" type="radio"/>	The locating of the page table using PTBR also involves paging translation
<input type="radio"/>	<input checked="" type="radio"/>	Size of page table is always determined by the size of RAM
<input checked="" type="radio"/>	<input type="radio"/>	The page table is itself present in Physical memory
<input checked="" type="radio"/>	<input type="radio"/>	Maximum Size of page table is determined by number of bits used for page number
<input checked="" type="radio"/>	<input type="radio"/>	The physical address may not be of the same size (in bits) as the logical address

The PTBR is present in the CPU as a register: True

The page table is indexed using frame number: False

The page table is indexed using page number: True

The locating of the page table using PTBR also involves paging translation: False

Size of page table is always determined by the size of RAM: False

The page table is itself present in Physical memory: True

Maximum Size of page table is determined by number of bits used for page number: True

The physical address may not be of the same size (in bits) as the logical address: True

Question 19

Correct

Mark 2.00 out of 2.00

Given below is shared memory code with two processes sharing a memory segment.

The first process sends a user input string to second process. The second capitalizes the string. Then the first process prints the capitalized version.

Fill in the blanks to complete the code.

// First process

```
#define SHMSZ 27

int main()
{
    char c;
    int shmid;
    key_t key;
    char *shm, *s, string[128];
    key = 5679;
    if ((shmid =
        shmget
        ✓ (key, SHMSZ, IPC_CREAT | 0666)) < 0) {
        perror("shmget");
        exit(1);
    }
    if ((shm =
        shmat
        ✓ (shmid, NULL, 0)) == (char *) -1) {
        perror("shmat");
        exit(1);
    }
    s = shm;
    *s = '$';
    scanf("%s", string);
    strcpy(s + 1, string);
    *s =
        @
        ✓ ';' //note the quotes
    while(*s != '
        $
        ')
        sleep(1);
        printf("%s\n", s + 1);
        exit(0);
}
```

//Second process

```
#define SHMSZ 27

int main()
{
    int shmid;
    key_t key;
    char *shm, *s;
    int i;
    char string[128];
    key =
        5679
```

```

✓ ;
if ((shmid = shmget(key, SHMSZ, 0666)) < 0) {
    perror("shmget");
    exit(1);
}
if ((shm = shmat(shmid, NULL, 0)) == (char *) -1) {
    perror("shmat");
    exit(1);
}
s =

✓ ;
while(*s != '@')
    sleep(1);
for(i = 0; i < strlen(s + 1); i++)
    s[i + 1] = toupper(s[i + 1]);
*s = '$';
exit(0);
}

```

Question 20

Partially correct

Mark 0.25 out of 0.50

Map the functionality/use with function/variable in xv6 code.

return a free page, if available; 0, otherwise

Create page table entries for a given range of virtual and physical addresses; including page directory entries if needed

Array listing the kernel memory mappings, to be used by setupkvm()

Setup kernel part of a page table, mapping kernel code, data, read-only data, I/O space, devices

Return address of page table entry in a given page directory, for a given virtual address; creates page table if necessary

Setup kernel part of a page table, and switch to that page table

 kinit1()

 mappages()

 kmap[]

 kvmalloc()

 walkpgdir()

 setupkvm()

Your answer is partially correct.

You have correctly selected 3.

The correct answer is: return a free page, if available; 0, otherwise → kalloc(), Create page table entries for a given range of virtual and physical addresses; including page directory entries if needed → mappages(), Array listing the kernel memory mappings, to be used by setupkvm() → kmap[], Setup kernel part of a page table, mapping kernel code, data, read-only data, I/O space, devices → setupkvm(), Return address of page table entry in a given page directory, for a given virtual address; creates page table if necessary → walkpgdir(), Setup kernel part of a page table, and switch to that page table → kvmalloc()

Question 21

Partially correct

Mark 1.53 out of 2.50

Order events in xv6 timer interrupt code

(Transition from process P1 to P2's code.)

P2 is selected and marked RUNNING

12 ✓

Change of stack from user stack to kernel stack of P1

3 ✓

Timer interrupt occurs

2 ✓

alltraps() will call iret

17 ✗

change to context of P2, P2's kernel stack in use now

13 ✓

P2's trap() will return to alltraps

16 ✗

jump in vector.S

4 ✓

P2 will return from sched() in yield()

14 ✗

yield() is called

8 ✓

trap() is called

7 ✓

Process P2 is executing

18 ✗

P1 is marked as RUNNABLE

9 ✓

P2's yield() will return in trap()

15 ✗

Process P1 is executing

1 ✓

sched() is called,

11 ✗

change to context of the scheduler, scheduler's stack in use now

10 ✗

jump to alltraps

5 ✓

Trapframe is built on kernel stack of P1

6 ✓

Your answer is partially correct.

You have correctly selected 11.

The correct answer is: P2 is selected and marked RUNNING → 12, Change of stack from user stack to kernel stack of P1 → 3, Timer interrupt occurs → 2, alltraps() will call iret → 18, change to context of P2, P2's kernel stack in use now → 13, P2's trap() will return to alltraps → 17, jump in vector.S → 4, P2 will return from sched() in yield() → 15, yield() is called → 8, trap() is called → 7, Process P2 is executing → 14, P1 is marked as RUNNABLE → 9, P2's yield() will return in trap() → 16, Process P1 is executing → 1, sched() is called, → 10, change to context of the scheduler, scheduler's stack in use now → 11, jump to alltraps → 5, Trapframe is built on kernel stack of P1 → 6

Question 22

Incorrect

Mark 0.00 out of 1.00

Given that the memory access time is 200 ns, probability of a page fault is 0.7 and page fault handling time is 8 ms,
The effective memory access time in nanoseconds is:

Answer: ✖

The correct answer is: 5600060.00

Question 23

Correct

Mark 0.25 out of 0.25

Select the state that is not possible after the given state, for a process:

- New: Running ✓
- Ready : Waiting ✓
- Running: None of these ✓
- Waiting: Running ✓

Question 24

Partially correct

Mark 0.63 out of 1.00

Select the correct statements about sched() and scheduler() in xv6 code

- a. scheduler() switches to the selected process's context ✓
- b. When either sched() or scheduler() is called, it does not return immediately to caller ✓
- c. After call to swtch() in sched(), the control moves to code in scheduler()
- d. Each call to sched() or scheduler() involves change of one stack inside swtch() ✓
- e. After call to swtch() in scheduler(), the control moves to code in sched()
- f. When either sched() or scheduler() is called, it results in a context switch ✓
- g. sched() switches to the scheduler's context ✓
- h. sched() and scheduler() are co-routines

Your answer is partially correct.

You have correctly selected 5.

The correct answers are: sched() and scheduler() are co-routines, When either sched() or scheduler() is called, it does not return immediately to caller, When either sched() or scheduler() is called, it results in a context switch, sched() switches to the scheduler's context, scheduler() switches to the selected process's context, After call to swtch() in scheduler(), the control moves to code in sched(), After call to swtch() in sched(), the control moves to code in scheduler(), Each call to sched() or scheduler() involves change of one stack inside swtch()

Question 25

Correct

Mark 0.25 out of 0.25

The data structure used in kalloc() and kfree() in xv6 is

- a. Doubly linked circular list
- b. Singly linked circular list
- c. Double linked NULL terminated list
- d. Singly linked NULL terminated list



Your answer is correct.

The correct answer is: Singly linked NULL terminated list

[◀ \(Assignment\) lseek system call in xv6](#)

Jump to...

Dashboard / My courses / Computer Engineering & IT / CEIT-Even-sem-20-21 / QS-Even-sem-2020-21 / 16 May - 22 May / End Sem Exam OS-2021

Started on Saturday, 22 May 2021, 8:00 AM

State Finished

Completed on Saturday, 22 May 2021, 9:30 AM

Time taken 1 hour 30 mins

Grade 26.12 out of 40.00 (65%)

Question 1

Incorrect

Mark 0.00 out of 1.00

A 4 GB disk with 1 KB of block size would require these many number of **blocks** for its free block bitmap:

Answer: 4096 ✖

The correct answer is: 512

Question 2

Correct

Mark 1.00 out of 1.00

Given that the memory access time is 110 ns, probability of a page fault is 0.5 and page fault handling time is 12 ms,

The effective memory access time in nanoseconds is:

Answer: 6000165 ✓

The correct answer is: 6000055.00

Question 3

Incorrect

Mark 0.00 out of 1.00

The maximum size of a file in number of blocks of BSIZE in xv6 code is

(write a number only)

Answer: 268 ✖

The correct answer is: 138

Question 4

Incorrect

Mark 0.00 out of 1.00

Calculate the average waiting time using

Round Robin scheduling with time quantum of 5 time units
for the following workload

assuming that they arrive in the order written below.

Process Burst Time

P1	5
P2	7
P3	6
P4	2

Write only a number in the answer upto two decimal points.

Answer: 40.75 ✖

The correct answer is: 10.25



Question 5

Correct

Mark 1.00 out of 1.00

For the reference string

4 2 5 1 0 1 2 5 4 1 2

the number of page faults, including initial ones,
with FIFO replacement and 2 frames are :

Answer: 10 ✓

4 -

4 2

5 2

5 1

0 1

-

2 1

2 5

4 5

4 1

2 1

The correct answer is: 10

Question 6

Correct

Mark 1.00 out of 1.00

Assuming a 16- KB page size, what is the page number for the address 428517 reference in decimal :

(give answer also in decimal)

Answer: 27 ✓

The correct answer is: 26



Question 7

Correct

Mark 1.00 out of 1.00

In the code below assume that each function can be executed concurrently by many threads/processes.
Ignore syntactical issues, and focus on the semantics.

This program is an example of

```
spinlock a, b; // assume initialized
thread1() {
    spinlock(b);
    //some code;
    spinlock(a);
    //some code;
    spinunlock(b);
    spinunlock(a);
}
thread2() {
    spinlock(a);
    //some code;
    spinlock(b);
    //some code;
    spinunlock(b);
    spinunlock(a);
}
```

- a. Deadlock ✓
- b. Self Deadlock
- c. None of these
- d. Deadlock or livelock depending on actual race
- e. Livelock

Your answer is correct.

The correct answer is: Deadlock



Question 8

Partially correct

Mark 1.33 out of 2.00

Match the snippets of xv6 code with the core functionality they achieve, or problems they avoid.
"..." means some code.

```
static inline uint
xchg(volatile uint *addr, uint newval)
{
    uint result;

    // The + in "+m" denotes a read-modify-write operand.
    asm volatile("lock; xchgl %0, %1" :
        "+m" ("*addr), "=a" (result) :
        "1" (newval) :
        "cc");
    return result;
}
```

Atomic compare and swap instruction (to be expanded inline into code)



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

If you don't do this, a process may be running on two processors parallelly



```
void
acquire(struct spinlock *lk)
{
    ...
    __sync_synchronize();
}
```

Tell compiler not to reorder memory access beyond this line



Your answer is partially correct.

You have correctly selected 2.

The correct answer is: static inline uint
xchg(volatile uint *addr, uint newval)

```
{
    uint result;
```

```
// The + in "+m" denotes a read-modify-write operand.
asm volatile("lock; xchgl %0, %1" :
    "+m" ("*addr), "=a" (result) :
    "1" (newval) :
    "cc");
return result;
} → Atomic compare and swap instruction (to be expanded inline into code), void
sleep(void *chan, struct spinlock *lk)
{
    ...
    if(lk != &ptable.lock){
        acquire(&ptable.lock);
        release(lk);
    } → Avoid a self-deadlock, void
    acquire(struct spinlock *lk)
{
    ...
    __sync_synchronize(); → Tell compiler not to reorder memory access beyond this line
```

Question 9

Correct

Mark 1.00 out of 1.00

Predict the output of the program given here.

Assume that all the path names for the programs are correct. For example "/usr/bin/echo" will actually run echo command.

Assume that there is no mixing of print output on screen if two of them run concurrently.

In the answer replace a new line by a single space.

For example::

good

output

should be written as good output

--

```
main() {  
    int i;  
    i = fork();  
    if(i == 0)  
        execl("/usr/bin/echo", "/usr/bin/echo", "hi", 0);  
    else  
        wait(0);  
    fork();  
    execl("/usr/bin/echo", "/usr/bin/echo", "one", 0);  
}
```

Answer: hi one one 

The correct answer is: hi one one

Question 10

Partially correct

Mark 1.67 out of 2.00

Select all the blocks that may need to be written back to disk (if updated, of-course), as "Yes", when an operation of deleting a file is carried out on ext2 file system.

An option has to be correct entirely to be marked "Yes"

Superblock

Yes 

One or multiple data blocks of the parent directory

No 

One or more data bitmap blocks for the parent directory

No 

Block bitmap(s) for all the blocks of the file

No 

Possibly one block bitmap corresponding to the parent directory

Yes 

Data blocks of the file

No 

Your answer is partially correct.

only one data block of parent directory. multiple blocks not possible. an entry is always contained within one single block

You have correctly selected 5.

The correct answer is: Superblock → Yes, One or multiple data blocks of the parent directory → No, One or more data bitmap blocks for the parent directory → No, Block bitmap(s) for all the blocks of the file → Yes, Possibly one block bitmap corresponding to the parent directory → Yes, Data blocks of the file → No

Question 11

Correct

Mark 1.00 out of 1.00

Select all the correct statements about bootloader.

Every wrong selection will deduct marks proportional to $1/n$ where n is total wrong choices in the question.

You will get minimum a zero.

- a. Modern Bootloaders often allow configuring the way an OS boots ✓
- b. Bootloaders allow selection of OS to boot from ✓
- c. Bootloader must be one sector in length
- d. The bootloader loads the BIOS
- e. LILO is a bootloader ✓

Your answer is correct.

The correct answers are: LILO is a bootloader, Modern Bootloaders often allow configuring the way an OS boots, Bootloaders allow selection of OS to boot from



Question 12

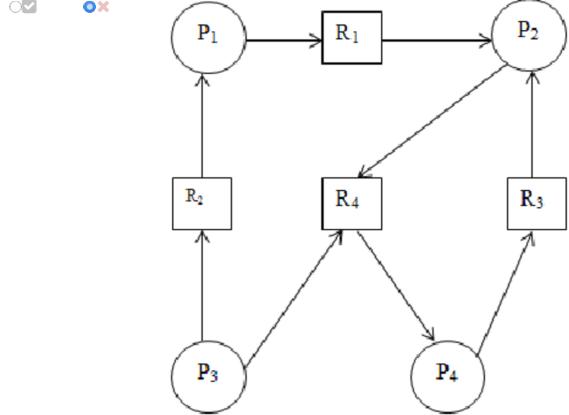
Incorrect

Mark 0.00 out of 1.00

For each of the resource allocation diagram shown,
infer whether the graph contains at least one deadlock or not.

Yes

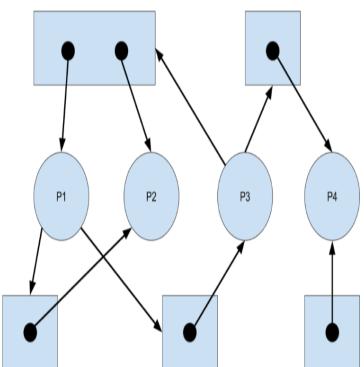
No



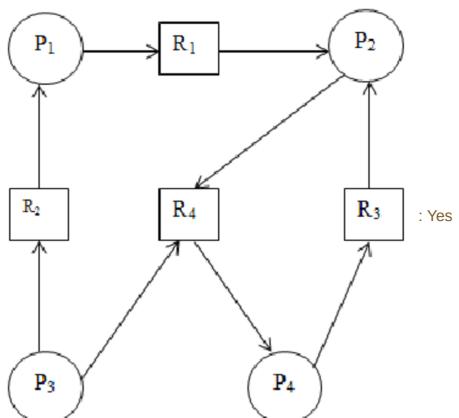
✗

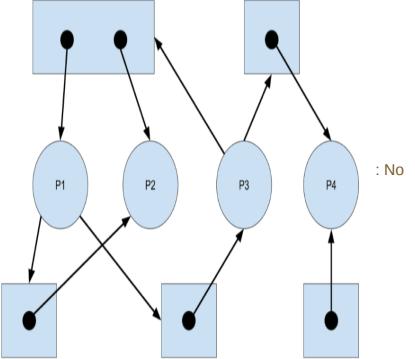
✗

✓



✗





Question 13

Partially correct

Mark 0.71 out of 1.00

Mark the statements about device drivers by marking as True or False.

True	False	
<input checked="" type="radio"/>	<input type="radio"/> ✘	It's possible that a particular hardware has multiple device drivers available for it.
<input checked="" type="radio"/>	<input type="radio"/> ✘	xv6 has device drivers for IDE disk and console.
<input checked="" type="radio"/>	<input type="radio"/> ✘	A disk driver converts OS's logical view of disk into physical locations on disk.
<input checked="" type="radio"/>	<input type="radio"/> ✘	A device driver code is specific to a hardware device
<input checked="" type="radio"/>	<input type="radio"/> ✘	All devices of the same type (e.g. 2 hard disks) can typically use the same device driver
<input checked="" type="radio"/>	<input type="radio"/> ✘	Writing a device driver mandatorily demands reading the technical documentation about the hardware.
<input type="radio"/> ✘	<input checked="" type="radio"/>	Device driver is an intermediary between the end-user and OS

It's possible that a particular hardware has multiple device drivers available for it.: True

xv6 has device drivers for IDE disk and console.: True

A disk driver converts OS's logical view of disk into physical locations on disk.: True

A device driver code is specific to a hardware device: True

All devices of the same type (e.g. 2 hard disks) can typically use the same device driver: True

Writing a device driver mandatorily demands reading the technical documentation about the hardware.: True

Device driver is an intermediary between the end-user and OS: False

Question 14

Partially correct

Mark 0.33 out of 1.00

Consider this program.

Some statements are identified using the // comment at the end.

Assume that `=` is an atomic operation.

```
#include <stdio.h>
#include <pthread.h>
long c = 0, c1 = 0, c2 = 0, run = 1;
void *thread1(void *arg) {
    while(run == 1) { //E
        c = 10; //A
        c1 = c2 + 5; //B
    }
}
void *thread2(void *arg) {
    while(run == 1) { //F
        c = 20; //C
        c2 = c1 + 3; //D
    }
}
int main() {
    pthread_t th1, th2;
    pthread_create(&th1, NULL, thread1, NULL);
    pthread_create(&th2, NULL, thread2, NULL);
    sleep(2);
    run = 0;
    printf(stdout, "c = %ld c1+c2 = %ld c1 = %ld c2 = %ld \n", c, c1+c2, c1, c2);
    fflush(stdout);
}
```

Which statements are part of the critical Section?

Yes	No	
<input checked="" type="radio"/> <input checked="" type="checkbox"/>	<input type="radio"/> <input checked="" type="checkbox"/>	F
<input checked="" type="radio"/> <input checked="" type="checkbox"/>	<input checked="" type="radio"/> <input type="checkbox"/>	D
<input checked="" type="radio"/> <input checked="" type="checkbox"/>	<input type="radio"/> <input checked="" type="checkbox"/>	C
<input checked="" type="radio"/> <input checked="" type="checkbox"/>	<input type="radio"/> <input checked="" type="checkbox"/>	A
<input checked="" type="radio"/> <input checked="" type="checkbox"/>	<input checked="" type="radio"/> <input type="checkbox"/>	B
<input checked="" type="radio"/> <input checked="" type="checkbox"/>	<input type="radio"/> <input checked="" type="checkbox"/>	E

F: No

D: Yes

C: No

A: No

B: Yes

E: No

Question 15

Partially correct

Mark 1.43 out of 2.00

Mark statements as T/F

All statements are in the context of preventing deadlocks.

True**False**

<input checked="" type="radio"/>	<input type="radio"/>	A process holding one resources and waiting for just one more resource can also be involved in a deadlock.	✓
<input type="radio"/>	<input checked="" type="radio"/>	If a resource allocation graph contains a cycle then there is a guarantee of a deadlock	✗
<input type="radio"/>	<input checked="" type="radio"/>	The lock ordering to be followed to avoid circular wait is a code in OS that checks for compliance with decided order	✗
<input checked="" type="radio"/>	<input type="radio"/>	Circular wait is avoided by enforcing a lock ordering	✓
<input checked="" type="radio"/>	<input type="radio"/>	Hold and wait means a thread/process holding some locks and waiting for acquiring some.	✓
<input checked="" type="radio"/>	<input type="radio"/>	Deadlock is possible if all the conditions are met at the same time: Mutual exclusion, hold and wait, no pre-emption, circular wait.	✓
<input checked="" type="radio"/>	<input type="radio"/>	Mutual exclusion is a necessary condition for deadlock because it brings in locks on which deadlock happens	✓

A process holding one resources and waiting for just one more resource can also be involved in a deadlock.: True

If a resource allocation graph contains a cycle then there is a guarantee of a deadlock: False

The lock ordering to be followed to avoid circular wait is a code in OS that checks for compliance with decided order: False

Circular wait is avoided by enforcing a lock ordering: True

Hold and wait means a thread/process holding some locks and waiting for acquiring some.: True

Deadlock is possible if all the conditions are met at the same time: Mutual exclusion, hold and wait, no pre-emption, circular wait.: True

Mutual exclusion is a necessary condition for deadlock because it brings in locks on which deadlock happens: True

Question 16

Correct

Mark 1.00 out of 1.00

Match the left side use(or non-use) of a synchronization primitive with the best option on the right side.

This is the smallest primitive made available in software, using the hardware provided atomic instructions

 spinlock ✓

This tool is useful for event-wait scenarios

 semaphore ✓

This tool is more useful on multiprocessor systems

 spinlock ✓

This tool is quite attractive in solving the main bounded buffer problem

 semaphore ✓

This tool is very useful for waiting for 'something'

 condition variables ✓

Your answer is correct.

The correct answer is: This is the smallest primitive made available in software, using the hardware provided atomic instructions → spinlock, This tool is useful for event-wait scenarios → semaphore, This tool is more useful on multiprocessor systems → spinlock, This tool is quite attractive in solving the main bounded buffer problem → semaphore, This tool is very useful for waiting for 'something' → condition variables

Question 17

Correct

Mark 1.00 out of 1.00

The permissions -rwx--x--x on a file mean

- a. The file can be read only by the owner
- b. 'cat' on the file by owner will not work
- c. 'cat' on the file by any user will work
- d. 'rm' on the file by any user will work
- e. The file can be executed by anyone
- f. The file can be written only by the owner



Your answer is correct.

The correct answers are: The file can be executed by anyone, The file can be read only by the owner, The file can be written only by the owner, 'rm' on the file by any user will work

Question 18

Incorrect

Mark 0.00 out of 1.00

Note: for this question you get full marks if you select all and only correct options, you get ZERO if at least one option is wrong or not selected.

Select all the correct statements about log structured file systems.

- a. a transaction is said to be committed when all operations are written to file system
- b. log may be kept on same block device or another block device
- c. file system recovery may end up losing data
- d. even if file systems followed immediate writes (i.e. non-delayed writes), it could still require recovery
- e. file system recovery recovers all the lost data



Your answer is incorrect.

The correct answers are: file system recovery may end up losing data, log may be kept on same block device or another block device, even if file systems followed immediate writes (i.e. non-delayed writes), it could still require recovery

Question 19

Incorrect

Mark 0.00 out of 1.00

Consider the structure of directory entry in ext2, as shown in this diagram.

	inode	rec_len	file_type	name_len	name
0	21	12	1	2	.
12	22	12	2	2	.
24	53	16	5	2	h o m e
40	67	28	3	2	u s r
52	0	16	7	1	o l d f i l e
68	34	12	4	2	s b i n

Select the correct statements about the directory entry in ext2 file system.

The correct formula for rec_len is (when entries are continuously stored)

- a. $\text{rec_len} = \text{sizeof(inode entry)} + \text{sizeof(name len entry)} + \text{sizeof(file type entry)} + (\text{strlen(name)} + (-1) * (\text{strlen(name)} \% 4))$
- b. $\text{rec_len} = \text{sizeof(inode entry)} + \text{sizeof(name len entry)} + \text{sizeof(file type entry)} + (\text{strlen(name)} + (\text{strlen(name)} - 4) \% 4)$
- c. $\text{rec_len} = \text{sizeof(inode entry)} + \text{sizeof(name len entry)} + \text{sizeof(file type entry)} + (\text{strlen(name)} + 4 - (\text{strlen(name)} \% 4))$ ✗
- d. $\text{rec_len} = \text{sizeof(inode entry)} + \text{sizeof(name len entry)} + \text{sizeof(file type entry)} + (\text{strlen(name)} + (-1) * (\text{strlen(name)} - 4))$
- e. $\text{rec_len} = \text{sizeof(inode entry)} + \text{sizeof(name len entry)} + \text{sizeof(file type entry)} + (\text{strlen(name)} \% 4)$
- f. $\text{rec_len} = \text{sizeof(inode entry)} + \text{sizeof(name len entry)} + \text{sizeof(file type entry)} + \text{strlen(name)}$

Your answer is incorrect.

The correct answer is: $\text{rec_len} = \text{sizeof(inode entry)} + \text{sizeof(name len entry)} + \text{sizeof(file type entry)} + (\text{strlen(name)} + (-1) * (\text{strlen(name)} - 4))$

Question 20

Partially correct

Mark 0.50 out of 1.00

Mark whether the given sequence of events is possible or not-possible. Also, select the reason for your answer.

For each sequence it's a not-possible sequence if some important event is not mentioned in the sequence.

Assume that the kernel code is non-interruptible and uniprocessor system.

Process P1 executing a system call
 Timer interrupt
 Generic interrupt handler runs
 Scheduler runs
 Scheduler selects P2 for execution
 P2 returns from timer interrupt handler
 Process p2, user code executing

This sequence of events is: ✓

Because

✗

Question 21

Incorrect

Mark 0.00 out of 1.00

The given semaphore implementation faces which problem?

Assume any suitable code for signal()

Note: blocks means waits in a wait queue.

```
struct semaphore {  
    int val;  
    spinlock lk;  
};  
sem_init(semaphore *s, int initval) {  
    s->val = initval;  
    s->sl = 0;  
}  
wait(semaphore *s) {  
    spinlock(&(s->sl));  
    while(s->val <=0)  
        ;  
    (s->val)--;  
    spinunlock(&(s->sl));  
}
```

- a. blocks holding a spinlock
- b. deadlock
- c. too much spinning, bounded wait not guaranteed ✖
- d. not holding lock after unblock

Your answer is incorrect.

The correct answer is: deadlock

Question 22

Partially correct

Mark 0.80 out of 1.00

Mark statements True/False w.r.t. change of states of a process.

Reference: The process state diagram (and your understanding of how kernel code works)

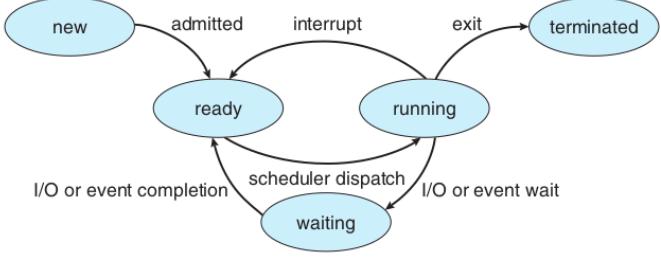


Figure 3.2 Diagram of process state.

True

False

<input type="radio"/> ✗	<input checked="" type="checkbox"/> ✗	A process in RUNNING state only can become TERMINATED because scheduler moves it to ZOMBIE state	✓
<input checked="" type="checkbox"/> ✗	<input type="radio"/> ✗	A process in READY state can not go to WAITING state because the resource on which it will WAIT will not be in use when process is in READY state.	✗
<input checked="" type="checkbox"/> ✗	<input type="radio"/> ✗	A process in WAITING state can not become RUNNING because the event it's waiting for has not occurred	✓
<input checked="" type="checkbox"/> ✗	<input type="radio"/> ✗	Every process has to go through ZOMBIE state, at least for a small duration.	✓
<input checked="" type="checkbox"/> ✗	<input type="radio"/> ✗	Only a process in READY state is considered by scheduler	✓

A process in RUNNING state only can become TERMINATED because scheduler moves it to ZOMBIE state: False

A process in READY state can not go to WAITING state because the resource on which it will WAIT will not be in use when process is in READY state.: False

A process in WAITING state can not become RUNNING because the event it's waiting for has not occurred: True

Every process has to go through ZOMBIE state, at least for a small duration.: True

Only a process in READY state is considered by scheduler: True

Question 23

Correct

Mark 1.00 out of 1.00

Select T/F for statements about Volume Managers.

Do pay attention to the use of the words physical partition and physical volume.

True**False**

<input checked="" type="radio"/>	<input type="radio"/> ✗	The volume manager can create further internal sub-divisions of a physical partition for efficiency or features.	✓
<input checked="" type="radio"/>	<input type="radio"/> ✗	A logical volume can be extended in size but upto the size of volume group	✓
<input checked="" type="radio"/>	<input type="radio"/> ✗	A logical volume may span across multiple physical volumes	✓
<input checked="" type="radio"/>	<input type="radio"/> ✗	The volume manager stores additional metadata on the physical disk partitions	✓
<input checked="" type="radio"/>	<input type="radio"/> ✗	A physical partition should be initialized as a physical volume, before it can be used by volume manager.	✓
<input checked="" type="radio"/>	<input type="radio"/> ✗	A volume group consists of multiple physical volumes	✓
<input checked="" type="radio"/>	<input type="radio"/> ✗	A logical volume may span across multiple physical partitions	✓ since a physical volume is made up of physical partitions, and a volume can span across multiple PVs, it can also span across multiple PP

The volume manager can create further internal sub-divisions of a physical partition for efficiency or features.: True

A logical volume can be extended in size but upto the size of volume group: True

A logical volume may span across multiple physical volumes: True

The volume manager stores additional metadata on the physical disk partitions: True

A physical partition should be initialized as a physical volume, before it can be used by volume manager.: True

A volume group consists of multiple physical volumes: True

A logical volume may span across multiple physical partitions: True

Question 24

Correct

Mark 1.00 out of 1.00

Map the block allocation scheme with the problem it suffers from

(Match pairs 1-1, match a scheme with the problem that it suffers from relatively the most, compared to others)

Continuous allocation	need for compaction	✓
Linked allocation	Too many seeks	✓
Indexed Allocation	Overhead of reading metadata blocks	✓

Your answer is correct.

The correct answer is: Continuous allocation → need for compaction, Linked allocation → Too many seeks, Indexed Allocation → Overhead of reading metadata blocks

Question 25

Correct

Mark 1.00 out of 1.00

This one is not a system call:

- a. open
- b. read
- c. write
- d. scheduler



Your answer is correct.

The correct answer is: scheduler



Question 26

Correct

Mark 1.00 out of 1.00

Match the pairs.

This question is based on your general knowledge about operating systems/related concepts and their features.

Java threads	monitors,re-entrant locks, semaphores	✓
Linux threads	atomic-instructions, spinlocks, etc.	✓
POSIX threads	semaphore, mutex, condition variables	✓

Your answer is correct.

The correct answer is: Java threads → monitors,re-entrant locks, semaphores, Linux threads → atomic-instructions, spinlocks, etc., POSIX threads → semaphore, mutex, condition variables

Question 27

Correct

Mark 1.00 out of 1.00

Consider the following list of free chunks, in continuous memory management:

7k, 15k, 21k, 14k, 19k, 6k

Suppose there is a request for chunk of size 5k, then the free chunk selected under each of the following schemes will be

Best fit:	6k	✓
First fit:	7k	✓
Worst fit:	21k	✓

Question 28

Correct

Mark 1.00 out of 1.00

This one is not a scheduling algorithm

- a. Round Robin
- b. SJF
- c. Mergesort
- d. FCFS



Your answer is correct.

The correct answer is: Mergesort

Question 29

Correct

Mark 1.00 out of 1.00

Mark whether the concept is related to scheduling or not.

Yes	No	
<input checked="" type="radio"/>	<input type="radio"/>	timer interrupt
<input checked="" type="radio"/>	<input type="radio"/>	context-switch
<input checked="" type="radio"/>	<input type="radio"/>	ready-queue
<input type="radio"/>	<input checked="" type="radio"/>	file-table
<input checked="" type="radio"/>	<input type="radio"/>	runnable process

timer interrupt: Yes

context-switch: Yes

ready-queue: Yes

file-table: No

runnable process: Yes



Question 30

Partially correct

Mark 1.00 out of 2.00

Map ext2 data structure features with their purpose

Many copies of Superblock Choose...**Free blocks count in superblock and group descriptor**

Redundancy to ensure the most crucial data structure is not lost

**Used directories count in group descriptor**

is redundant and helps do calculations of directory entries faster

**Combining file type and access rights in one variable**

saves 1 byte of space

**rec_len field in directory entry**

Try to keep all the data of a directory and its file close together in a group

**File Name is padded**

aligns all memory accesses on word boundary, improving performance

**Inode bitmap is one block**

limits total number of files that can belong to a group

**Block bitmap is one block**

Limits the size of a block group, thus improvising on purpose of a group

**Mount count in superblock**

to enforce file check after certain amount of mounts at boot time

**Inode table location in Group Descriptor**

is redundant and helps do calculations of directory entries faster

**Inode table**

All inodes are kept together so that one disk read leads to reading many inodes together, effectively doing a buffering of subsequent inode reads, and to save space on disk

**A group**

Redundancy to ensure the most crucial data structure is not lost



Your answer is partially correct.

You have correctly selected 6.

The correct answer is: **Many copies of Superblock** → Redundancy to ensure the most crucial data structure is not lost, **Free blocks count in superblock and group descriptor** → Redundancy to help fsck restore consistency, **Used directories count in group descriptor** → attempt is made to evenly spread the first-level directories, this count is used there, **Combining file type and access rights in one variable** → saves 1 byte of space, **rec_len field in directory entry** → allows holes and linking of entries in directory, File Name is padded → aligns all memory accesses on word boundary, improving performance, **Inode bitmap is one block** → limits total number of files that can belong to a group, **Block bitmap is one block** → Limits the size of a block group, thus improvising on purpose of a group, **Mount count in superblock** → to enforce file check after certain amount of mounts at boot time, **Inode table location in Group Descriptor** → Obvious, as it's per group and not per file-system, **Inode table** → All inodes are kept together so that one disk read leads to reading many inodes together, effectively doing a buffering of subsequent inode reads, and to save space on disk, **A group** → Try to keep all the data of a directory and its file close together in a group

Question 31

Partially correct

Mark 1.85 out of 2.00

Mark True/False

Statements about scheduling and scheduling algorithms

True	False	
<input checked="" type="radio"/>	<input type="radio"/>	The nice() system call is used to set priorities for processes
<input checked="" type="radio"/>	<input type="radio"/>	Aging is used to ensure that low-priority processes do not starve in priority scheduling.
<input type="radio"/>	<input checked="" type="radio"/>	In non-pre-emptive priority scheduling, the highest priority process is scheduled and runs until it gives up CPU.
<input checked="" type="radio"/>	<input type="radio"/>	xv6 code does not care about Processor Affinity
<input checked="" type="radio"/>	<input type="radio"/>	In pre-emptive priority scheduling, priority is implemented by assigning more time quantum to higher priority process.
<input checked="" type="radio"/>	<input type="radio"/>	A scheduling algorithm is non-preemptive if it does context switch only if a process voluntarily relinquishes CPU or it terminates.
<input checked="" type="radio"/>	<input type="radio"/>	Processor Affinity refers to memory accesses of a process being stored on cache of that processor
<input checked="" type="radio"/>	<input type="radio"/>	Response time will be quite poor on non-interruptible kernels
<input checked="" type="radio"/>	<input type="radio"/>	Shortest Remaining Time First algorithm is nothing but pre-emptive Shortest Job First algorithm
<input checked="" type="radio"/>	<input type="radio"/>	On Linuxes the CPU utilisation is measured as the time spent in scheduling the idle thread
<input checked="" type="radio"/>	<input type="radio"/>	Generally the voluntary context switches are much more than non-voluntary context switches on a Linux system.
<input checked="" type="radio"/>	<input type="radio"/>	Pre-emptive scheduling leads to many race conditions in kernel code.
<input checked="" type="radio"/>	<input type="radio"/>	Statistical observations tell us that most processes have large number of small CPU bursts and relatively smaller numbers of large CPU bursts.

The nice() system call is used to set priorities for processes.: True

Aging is used to ensure that low-priority processes do not starve in priority scheduling.: True

In non-pre-emptive priority scheduling, the highest priority process is scheduled and runs until it gives up CPU.: True

xv6 code does not care about Processor Affinity: True

In pre-emptive priority scheduling, priority is implemented by assigning more time quantum to higher priority process.: True

A scheduling algorithm is non-preemptive if it does context switch only if a process voluntarily relinquishes CPU or it terminates.: True

Processor Affinity refers to memory accesses of a process being stored on cache of that processor: True

Response time will be quite poor on non-interruptible kernels: True

Shortest Remaining Time First algorithm is nothing but pre-emptive Shortest Job First algorithm: True

On Linuxes the CPU utilisation is measured as the time spent in scheduling the idle thread: True

Generally the voluntary context switches are much more than non-voluntary context switches on a Linux system.: True

Pre-emptive scheduling leads to many race conditions in kernel code.: True

Statistical observations tell us that most processes have large number of small CPU bursts and relatively smaller numbers of large CPU bursts.: True

Question 32

Partially correct

Mark 1.17 out of 2.00

The unix file semantics demand that changes to any open file are visible immediately to any other processes accessing that file at that point in time.

Select the data-structure/programmatic features that ensure the implementation of unix semantics. (Assume there is no mmap())

Yes	No	
<input type="radio"/> <input checked="" type="checkbox"/>	All processes accessing the same file share the file descriptor among themselves	✓
<input type="radio"/> <input checked="" type="checkbox"/>	The pointer entry in the file descriptor array entry points to the data of the file directly	✓
<input checked="" type="checkbox"/> <input type="radio"/>	There is only one global file structure per on-disk file.	✗
<input type="radio"/> <input checked="" type="checkbox"/>	All file accesses are made using only global variables	✓
<input checked="" type="checkbox"/> <input type="radio"/>	The 'file offset' is shared among all the processes that access the file.	✗
<input type="radio"/> <input checked="" type="checkbox"/>	No synchronization is implemented so that changes are made available immediately.	✓
<input checked="" type="checkbox"/> <input type="radio"/>	A single spinlock is to be used to protect the unique global 'file structure' representing the file, thus synchronizing access, and making other processes wait for earlier process to finish writing so that writes get visible immediately.	✗
<input checked="" type="checkbox"/> <input type="radio"/>	There is only one in-memory copy of the on disk file's contents in kernel memory/buffers	✓
<input checked="" type="checkbox"/> <input type="radio"/>	The file descriptors in every PCB are pointers to the same global file structure.	✗
<input type="radio"/> <input checked="" type="checkbox"/>	The file descriptor array is external to PCB and all processes that share a file, have pointers to same file-descriptors' array	✓
<input checked="" type="checkbox"/> <input type="radio"/>	All file structures representing any open file, give access to the same in-memory copy of the file's contents	✓
<input checked="" type="checkbox"/> <input type="radio"/>	The 'file offset' index is stored outside the file-structure to which file-descriptor array points	✗

All processes accessing the same file share the file descriptor among themselves: No

The pointer entry in the file descriptor array entry points to the data of the file directly: No

There is only one global file structure per on-disk file.: No

All file accesses are made using only global variables: No

The 'file offset' is shared among all the processes that access the file.: No

No synchronization is implemented so that changes are made available immediately.: No

A single spinlock is to be used to protect the unique global 'file structure' representing the file, thus synchronizing access, and making other processes wait for earlier process to finish writing so that writes get visible immediately.: No

There is only one in-memory copy of the on disk file's contents in kernel memory/buffers: Yes

The file descriptors in every PCB are pointers to the same global file structure.: No

The file descriptor array is external to PCB and all processes that share a file, have pointers to same file-descriptors' array: No

All file structures representing any open file, give access to the same in-memory copy of the file's contents: Yes

The 'file offset' index is stored outside the file-structure to which file-descriptor array points: No

Question 33

Partially correct

Mark 0.33 out of 2.00

Map the function in xv6's file system code, to its perceived logical layer.

namei	inode	✗
filestat()	Choose...	
dirlookup	directory	✓
ialloc	file descriptor	✗
stati	Choose...	
ideintr	buffer cache	✗
bread	Choose...	
balloc	file descriptor	✗
sys_chdir()	system call	✓
skipelem	system call	✗
commit	system call	✗
bmap	system call	✗

Your answer is partially correct.

You have correctly selected 2.

The correct answer is: namei → pathname lookup, filestat() → file descriptor, dirlookup → directory, ialloc → inode, stati → inode, ideintr → disk driver, bread → buffer cache, balloc → block allocation on disk, sys_chdir() → system call, skipelem → pathname lookup, commit → logging, bmap → inode

[◀ Course Exit Feedback](#)[Jump to...](#)[xv6-public-master ►](#)

Started on Saturday, 30 April 2022, 2:05:41 PM

State Finished

Completed on Saturday, 30 April 2022, 6:31:46 PM

Time taken 4 hours 26 mins

Grade 30.01 out of 40.00 (75%)

Question 1

Correct

Mark 1.00 out of 1.00

Predict the output of the program given here.

Assume that there is no mixing of printf output on screen if two of them run concurrently.

In the answer replace a new line by a single space.

For example::

good

output

should be written as good output

--

```
int main() {  
    int pid;  
    printf("hi\n");  
    pid = fork();  
    if(pid == 0) {  
        exit(0);  
    }  
    printf("bye\n");  
    fork();  
    printf("ok\n");  
}
```

Answer: hi bye ok ok



The correct answer is: hi bye ok ok

Question 2

Partially correct

Mark 1.25 out of 2.00

Match the snippets of xv6 code with the core functionality they achieve, or problems they avoid.

"..." means some code.

```
void  
panic(char *s)  
{  
...  
panicked = 1;
```

Disable interrupts to avoid another process's pointer being returned



```
void  
yield(void)  
{  
...  
release(&ptable.lock);  
}
```

Release the lock held by some another process



```
void  
acquire(struct spinlock *lk)  
{  
...  
_sync_synchronize();
```

Tell compiler not to reorder memory access beyond this line



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

If you don't do this, a process may be running on two processors parallelly



Disable interrupts to avoid deadlocks



```
struct proc*
myproc(void) {
...
pushcli();
c = mycpu();
p = c->proc;
popcli();
...
}
```

Avoid a self-deadlock



```
static inline uint
xchg(volatile uint *addr, uint newval)
{
    uint result;

    // The + in "+m" denotes a read-modify-write
    // operand.
    asm volatile("lock; xchgl %0, %1" :
        "+m" (*addr), "=a" (result) :
        "1" (newval) :
        "cc");
    return result;
}

void
acquire(struct spinlock *lk)
{
...
getcallerpcs(&lk, lk->pcs);
```

Atomic compare and swap instruction (to be expanded inline into code)



Traverse ebp chain to get sequence of instructions followed in functions calls



Your answer is partially correct.

You have correctly selected 5.

The correct answer is: void

```
panic(char *s)
{
...
panicked = 1; → Ensure that no printing happens on other processors, void
yield(void)
{
...
release(&ptable.lock);
}
```

→ Release the lock held by some another process, void

```
acquire(struct spinlock *lk)
{
...
__sync_synchronize();
```

→ Tell compiler not to reorder memory access beyond this line, `void`

```
sleep(void *chan, struct spinlock *lk)
```

```
{
```

```
...
```

```
if(lk != &ptable.lock){
```

```
    acquire(&ptable.lock);
```

```
    release(lk);
```

} → Avoid a self-deadlock, `void`

```
acquire(struct spinlock *lk)
```

```
{
```

```
    pushcli();
```

→ Disable interrupts to avoid deadlocks, `struct proc*`

```
myproc(void) {
```

```
...
```

```
    pushcli();
```

```
    c = mycpu();
```

```
    p = c->proc;
```

```
    popcli();
```

```
...
```

```
}
```

→ Disable interrupts to avoid another process's pointer being returned, `static inline uint`

```
xchg(volatile uint *addr, uint newval)
```

```
{
```

```
    uint result;
```

// The + in "+m" denotes a read-modify-write operand.

```
asm volatile("lock; xchgl %0, %1" :
```

```
    "+m" (*addr), "=a" (result) :
```

```
    "1" (newval) :
```

```
    "cc");
```

```
    return result;
```

} → Atomic compare and swap instruction (to be expanded inline into code), `void`

```
acquire(struct spinlock *lk)
```

```
{
```

```
...
```

```
    getcallerpcs(&lk, lk->pcs);
```

→ Traverse ebp chain to get sequence of instructions followed in functions calls

Question 3

Correct

Mark 1.00 out of 1.00

Which one of the following is not a system call

- a. mount
- b. lseek
- c. open
- d. lock



The correct answer is: lock

Question 4

Partially correct

Mark 1.13 out of 1.50

The following processes are being scheduled using a pre-emptive, priority-based, round-robin scheduling algorithm.

<u>Process</u>	<u>Priority</u>	<u>Burst</u>	<u>Arrival</u>
P_1	8	15	0
P_2	3	20	0
P_3	4	20	20
P_4	4	20	25
P_5	5	5	45
P_6	5	15	55

Each process is assigned a numerical priority, with a higher number indicating a higher relative priority. The scheduler will execute the currently highest priority process for its full duration, unless it gets pre-empted by newly arriving higher priority process. For processes with the same priority, a round-robin scheduler will be used with a time quantum of 10 units. If a process is pre-empted by a higher-priority process, the pre-empted process is placed at the front of the queue of same priority processes (so that its turn continues when higher priority process is over).

The order in which the processes get scheduled is (write your answer without a space, e.g. P1,P2,P3,P4,P5) :

P1,P2,P3,P4,P3,P5,P3,P6,P2



The turn-around time for the process P2 is:

85



The turn-around time for the process P6 is:

15



The process P5 finishes at time unit:

50



0-15 P1

15-20 P2

20-40 P3

40-45 P4

45-50 P5

50-55 P4

55-70 P6

70-80 P4

80-95 P2

--

P2 turnaround time = 95 - 15 = 80

Question 5

Partially correct

Mark 0.33 out of 1.00

Select all the correct statements about Shell

- a. Examples of shell are: bash, ksh, csh, zsh, sh, etc. ✓
- b. Shell converts the application code into system calls. ✗
- c. The essential job of the shell is to fork-exec the specified application ✓
- d. Examples of shell are: bash, ksh, csh, msh, ooosh, nosh, etc.
- e. The default shell for a user is specified in /etc/passwd on typical GNU/Linux systems.
- f. Shell is a layer on top of hardware

The correct answers are: The essential job of the shell is to fork-exec the specified application, Examples of shell are: bash, ksh, csh, zsh, sh, etc., The default shell for a user is specified in /etc/passwd on typical GNU/Linux systems.

Question 6

Correct

Mark 1.00 out of 1.00

In an ext2 file system, if the block size is 4KB and partition size is 64 GB, then the number of block groups will be:

Answer: ✓

$\text{size} * 1024 * 1024 / 4 \rightarrow \text{no of blocks}$

$\text{each group} = 8 * 4 * 1024 \text{ blocks} = 32768 \text{ blocks}$

$\text{so } \text{size} * 1024 * 1024 / (4 * 32768) \text{ number of groups}$

The correct answer is: 512.00

Question 7

Correct

Mark 1.00 out of 1.00

Calculate the average waiting time using
FCFS scheduling
for the following workload
assuming that they arrive in this order during the first time unit:

Process Burst Time

P1	2
P2	6
P3	2
P4	3

Write only a number in the answer upto two decimal points.

Answer: 5.00



P2 waits for 2 units

P3 waits for 2+6 units

P4 waits for 2 + 6 + 2 units of time

Total waiting = $2 + 2 + 6 + 2 + 6 + 2 = 20$ unitsAverage waiting time = $20/4 = 5$

The correct answer is: 5

Question 8

Correct

Mark 1.00 out of 1.00

Not a FOSS: Which of the following is/are not a FOSS operating system ?

- a. BSD Unix
- b. Darwin (core kernel of Mac-OS)
- c. GNU/Linux
- d. FreeBSD
- e. Minix
- f. Windows
- g. Open Solaris
- h. xv6



The correct answer is: Windows

Question 9

Incorrect

Mark 0.00 out of 1.00

Will this code work for a spinlock() operation? The intention here is to call compare-and-swap() only if the lock is not held (the if condition checks for the same).

```
void spinlock(int *lock) {  
    {  
        while (true) {  
            if (*lock == 0) {  
                /* lock appears to be available */  
                if (!compare_and_swap(lock, 0, 1))  
                    break  
            }  
        }  
    }  
}
```

- a. No, because this breaks the atomicity requirement of compare-and-test. ✖
- b. Yes, because there is no race to update the lock variable
- c. Yes, because no matter in which order the if-check and compare-and-swap run in multiple processes, only one process will succeed in compare-and-swap() and others will keep looping in while-loop.
- d. No, because in the case of both processes succeeding in the "if" condition, both may end up acquiring the lock.

Your answer is incorrect.

The correct answer is: Yes, because no matter in which order the if-check and compare-and-swap run in multiple processes, only one process will succeed in compare-and-swap() and others will keep looping in while-loop.

Question 10

Correct

Mark 1.00 out of 1.00

The following diagram that explains the concept of multi-threading

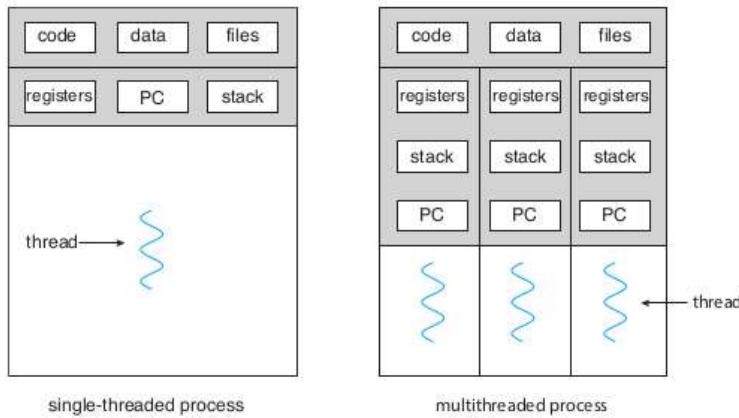


Figure 4.1 Single-threaded and multithreaded processes.

Leads to the following conclusions about implementation of threads

True	False	
<input type="radio"/> ✗	<input checked="" type="radio"/> ✗	A multi-threaded program can be split in multiple ELF files.
<input checked="" type="radio"/> ✗	<input type="radio"/> ✗	The memory management for the process in the kernel should not (most typically) change due to creation of a thread.
<input checked="" type="radio"/> ✗	<input type="radio"/> ✗	Each thread should be represented by a function that starts execution on a separate stack.
<input checked="" type="radio"/> ✗	<input type="radio"/> ✗	Switching between threads requires a "context switch" with change of registers involved in it

A multi-threaded program can be split in multiple ELF files.: False

The memory management for the process in the kernel should not (most typically) change due to creation of a thread.: True

Each thread should be represented by a function that starts execution on a separate stack.: True

Switching between threads requires a "context switch" with change of registers involved in it: True

Question 11

Correct

Mark 1.00 out of 1.00

Match each suggested semaphore implementation (discussed in class)

with the problems that it faces

```
struct semaphore {  
    int val;  
    spinlock lk;  
};  
sem_init(semaphore *s, int initval) {  
    s->val = initval;  
    s->sl = 0;  
}  
wait(semaphore *s) {  
    spinlock(&(s->sl));  
    while(s->val <=0)  
    ;  
    (s->val)--;  
    spinunlock(&(s->sl));  
}
```

deadlock



```
struct semaphore {  
    int val;  
    spinlock lk;  
    list l;  
};  
sem_init(semaphore *s, int initval) {  
    s->val = initval;  
    s->sl = 0;  
}  
block(semaphore *s) {  
    listappend(s->l, current);  
    schedule();  
}  
wait(semaphore *s) {  
    spinlock(&(s->sl));  
    while(s->val <=0) {  
        block(s);  
    }  
    (s->val)--;  
    spinunlock(&(s->sl));  
}
```

blocks holding a spinlock



```

struct semaphore {
    int val;
    spinlock lk;
    list l;
};

sem_init(semaphore *s, int initval) {
    s->val = initval;
    s->sl = 0;
}

block(semaphore *s) {
    listappend(s->l, current);
    spinunlock(&(s->sl));
    schedule();
}

wait(semaphore *s) {
    spinlock(&(s->sl));
    while(s->val <=0) {
        block(s);
    }
    (s->val)--;
    spinunlock(&(s->sl));
}

signal(semaphore *s) {
    spinlock(*(s->sl));
    (s->val)++;
    x = dequeue(s->sl) and enqueue(readyq, x);
    spinunlock(*(s->sl));
}

```

not holding lock after unblock



```

struct semaphore {
    int val;
    spinlock lk;
};

sem_init(semaphore *s, int initval) {
    s->val = initval;
    s->sl = 0;
}

wait(semaphore *s) {
    spinlock(&(s->sl));
    while(s->val <=0) {
        spinunlock(&(s->sl));
        spinlock(&(s->sl));
    }
    (s->val)--;
    spinunlock(&(s->sl));
}

```

too much spinning, bounded wait not guaranteed



Your answer is correct.

The correct answer is:

```

struct semaphore {
    int val;
    spinlock lk;
};

sem_init(semaphore *s, int initval) {
    s->val = initval;
    s->sl = 0;
}

wait(semaphore *s) {
    spinlock(&(s->sl));
    while(s->val <=0)
    ;
    (s->val)--;
    spinunlock(&(s->sl));
}

```

→ deadlock,

```
struct semaphore {
    int val;
    spinlock lk;
    list l;
};

sem_init(semaphore *s, int initval) {
    s->val = initval;
    s->sl = 0;
}

block(semaphore *s) {
    listappend(s->l, current);
    schedule();
}

wait(semaphore *s) {
    spinlock(&(s->sl));
    while(s->val <=0) {
        block(s);
    }
    (s->val)--;
    spinunlock(&(s->sl));
}
```

→ blocks holding a spinlock,

```
struct semaphore {
    int val;
    spinlock lk;
    list l;
};

sem_init(semaphore *s, int initval) {
    s->val = initval;
    s->sl = 0;
}

block(semaphore *s) {
    listappend(s->l, current);
    spinunlock(&(s->sl));
    schedule();
}

wait(semaphore *s) {
    spinlock(&(s->sl));
    while(s->val <=0) {
        block(s);
    }
    (s->val)--;
    spinunlock(&(s->sl));
}

signal(seamphore *s) {
    spinlock(*(&s->sl));
    (s->val)++;
    x = dequeue(s->sl) and enqueue(readyq, x);
    spinunlock(*(&s->sl));
}
```

→ not holding lock after unblock,

```
struct semaphore {
    int val;
    spinlock lk;
};

sem_init(semaphore *s, int initval) {
    s->val = initval;
    s->sl = 0;
}

wait(semaphore *s) {
    spinlock(&(s->sl));
    while(s->val <= 0) {
        spinunlock(&(s->sl));
        spinlock(&(s->sl));
    }
    (s->val)--;
    spinunlock(&(s->sl));
}
```

→ too much spinning, bounded wait not guaranteed

Question 12

Partially correct

Mark 1.85 out of 3.00

Suppose you are required to implement the priority scheduling algorithm in xv6. Select all the options that correctly reflect the changes that are required to be done in code:

Needed/Optional	Not Needed	
<input checked="" type="radio"/>	<input type="radio"/> <input checked="" type="checkbox"/>	Add a system call (like nice()) that allows to set priority of a process
<input checked="" type="radio"/>	<input checked="" type="radio"/> <input type="checkbox"/>	Set the priority of the new process to the default(using inheritance) during fork()
<input type="checkbox"/> <input checked="" type="checkbox"/>	<input checked="" type="radio"/>	Change yield() to re-set the timer value to zero for the process
<input checked="" type="radio"/>	<input type="radio"/> <input checked="" type="checkbox"/>	Insert code in scheduler() after if(p->state != RUNNABLE); continue; by a code that selects the highest priority process.
<input checked="" type="radio"/>	<input type="radio"/> <input checked="" type="checkbox"/>	Add a priority field to the struct proc
<input checked="" type="radio"/>	<input checked="" type="radio"/> <input type="checkbox"/>	Optionally remove the default timer value of 10000000 in lapticinit()
<input type="checkbox"/> <input checked="" type="checkbox"/>	<input type="radio"/> <input checked="" type="checkbox"/>	Change scheduler() to calculate the priority of the process using user specified value
<input type="checkbox"/> <input checked="" type="checkbox"/>	<input type="radio"/> <input checked="" type="checkbox"/>	Change exec() to add a priority to the process.
<input checked="" type="radio"/>	<input checked="" type="radio"/> <input type="checkbox"/>	Set a new value for the timer before you call swtch() in scheduler()
<input type="checkbox"/> <input checked="" type="checkbox"/>	<input checked="" type="radio"/> <input type="checkbox"/>	Change swtch() to set the timer value for the process
<input checked="" type="radio"/>	<input type="radio"/> <input checked="" type="checkbox"/>	The nice() system call needs to acquire the ptable.lock for setting the priority.
<input type="checkbox"/> <input checked="" type="checkbox"/>	<input checked="" type="radio"/> <input type="checkbox"/>	Set the priority of the process as per information in ELF file
<input type="checkbox"/> <input checked="" type="checkbox"/>	<input checked="" type="radio"/> <input type="checkbox"/>	Change sched() to set the priority of the process.

Add a system call (like nice()) that allows to set priority of a process: Needed/Optional

Set the priority of the new process to the default(using inheritance) during fork(): Needed/Optional

Change yield() to re-set the timer value to zero for the process: Not Needed

Insert code in scheduler() after if(p->state != RUNNABLE); continue; by a code that selects the highest priority process.: Needed/Optional

Add a priority field to the struct proc: Needed/Optional

Optionally remove the default timer value of 10000000 in lapicinit(): Needed/Optional

Change scheduler() to calculate the priority of the process using user specified value: Not Needed

Change exec() to add a priority to the process.: Not Needed

Set a new value for the timer before you call swtch() in scheduler(): Needed/Optional

Change swtch() to set the timer value for the process: Not Needed

The nice() system call needs to acquire the ptable.lock for setting the priority.: Needed/Optional

Set the priority of the process as per information in ELF file: Not Needed

Change sched() to set the priority of the process.: Not Needed

Question 13

Partially correct

Mark 0.75 out of 1.00

It is proposed that when a process does an illegal memory access, xv6 terminate the process by printing the error message "Illegal Memory Access". Select all the changes that need to be done to xv6 for this as True (Note that the changes proposed here may not cover the exhaustive list of all changes required) and the un-necessary/wrong changes as False.

Required	Un-necessary/Wrong	
<input checked="" type="radio"/>	<input type="radio"/>	Change exec to treat text/data sections separately and call allocuvm() with proper flags for page table entries
<input type="radio"/>	<input checked="" type="radio"/>	Add code that checks if the illegal memory access trap was due to an actual illegal memory access.
<input checked="" type="radio"/>	<input type="radio"/>	Change in the Makefile and instruct cc/ld to start the code of each program at some address other than 0
<input checked="" type="radio"/>	<input type="radio"/>	Handle the Illegal memory access trap in trap() function, and terminate the currently running process.
<input type="radio"/>	<input checked="" type="radio"/>	Mark each page as readonly in the page table mappings
<input checked="" type="radio"/>	<input type="radio"/>	Ensure that the address 0 is mapped to invalid
<input type="radio"/>	<input checked="" type="radio"/>	Change mappages() to set specified permissions on each page table entry
<input checked="" type="radio"/>	<input type="radio"/>	Change allocuvm() to call mappages() with proper permissions on each page table entry

Change exec to treat text/data sections separately and call allocuvm() with proper flags for page table entries: Required

Add code that checks if the illegal memory access trap was due to an actual illegal memory access.: Un-necessary/Wrong

Change in the Makefile and instruct cc/ld to start the code of each program at some address other than 0: Required

Handle the Illegal memory access trap in trap() function, and terminate the currently running process.: Required

Mark each page as readonly in the page table mappings: Un-necessary/Wrong

Ensure that the address 0 is mapped to invalid: Required

Change mappages() to set specified permissions on each page table entry: Un-necessary/Wrong

Change allocuvm() to call mappages() with proper permissions on each page table entry: Required

Question 14

Correct

Mark 1.00 out of 1.00

Map each signal with it's meaning

SIGSEGV	Invalid Memory Reference	✓
SIGPIPE	Broken Pipe	✓
SIGUSR1	User Defined Signal	✓
SIGCHLD	Child Stopped or Terminated	✓
SIGALRM	Timer Signal from alarm()	✓

The correct answer is: SIGSEGV → Invalid Memory Reference, SIGPIPE → Broken Pipe, SIGUSR1 → User Defined Signal, SIGCHLD → Child Stopped or Terminated, SIGALRM → Timer Signal from alarm()

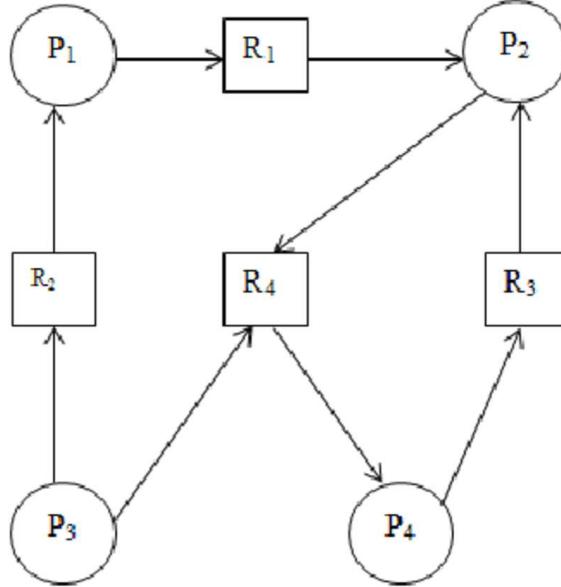
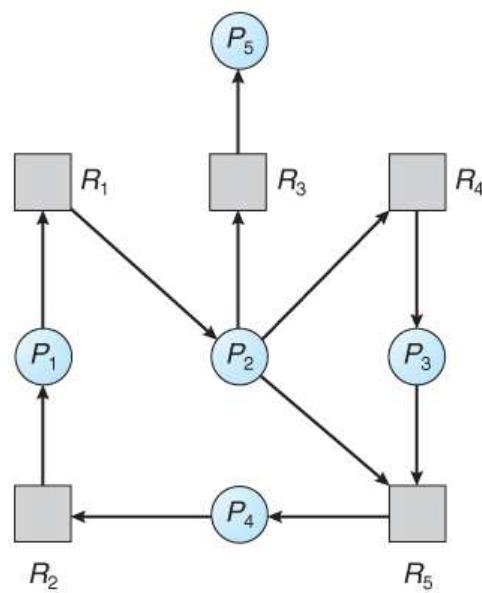
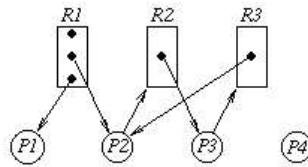
Question 15

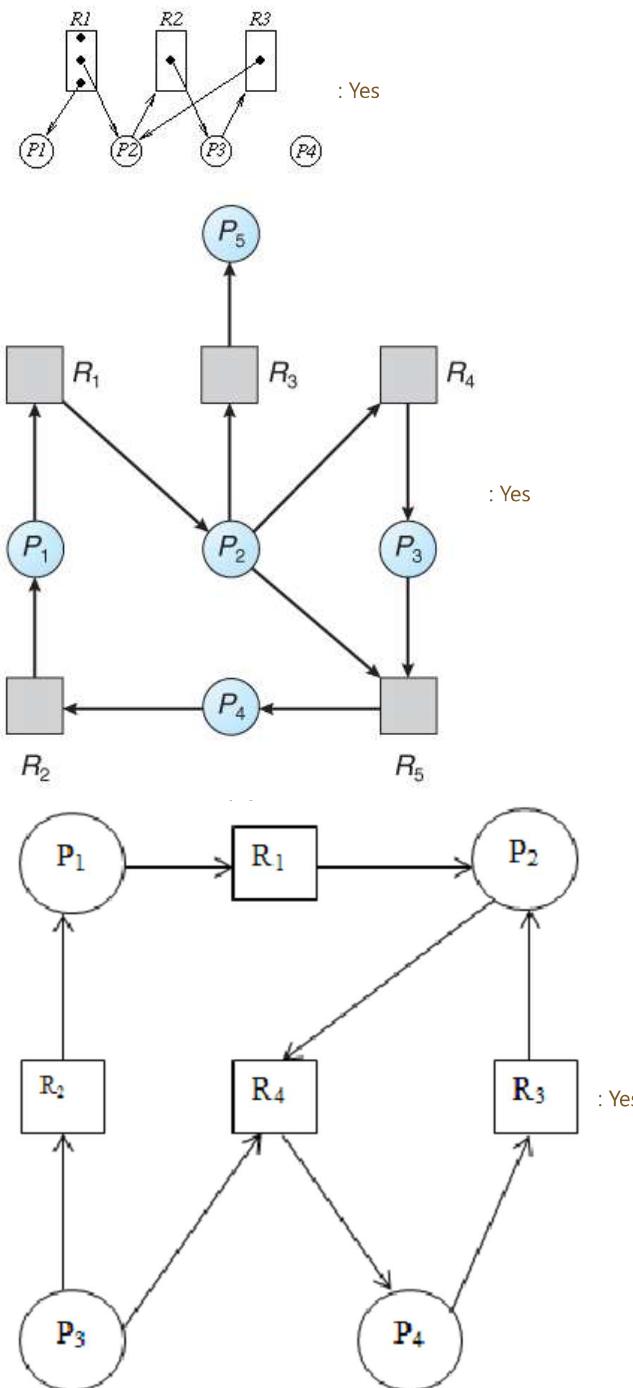
Correct

Mark 1.00 out of 1.00

For each of the resource allocation diagram shown,

infer whether the graph contains at least one deadlock or not.

Yes**No**



Question 16

Partially correct

Mark 0.67 out of 1.00

Match the pair

Hashed page table	Linear search on collision done by OS (e.g. SPARC Solaris) typically	✓
Inverted Page table	Linear/Parallel search using frame number in page table	✗
Hierarchical Paging	More memory access time per hierarchy	✓

Your answer is partially correct.

You have correctly selected 2.

The correct answer is: Hashed page table → Linear search on collision done by OS (e.g. SPARC Solaris) typically, Inverted Page table → Linear/Parallel search using page number in page table, Hierarchical Paging → More memory access time per hierarchy

Question 17

Correct

Mark 0.50 out of 0.50

Doing a lookup on the pathname /a/b/b/c/d for opening the file "d" requires reading ✓ no. of inodes. Assume that there are no hard/soft links on the path.

Write the answer as a number.

The correct answer is: 6

Question 18

Partially correct

Mark 0.40 out of 0.50

Which of the following statements are correct about xv6 and multiprocessor support ?

- a. xv6 supports only SMP ✓
- b. xv6 supports a variable number of processors (upto 8) and adjusts it's data structures according to number of processors
- c. Each processor on xv6 starts code execution in mpenter() when first processor sets "started" to 1. ✓
- d. In xv6 on x86, the first processor configures other processors in mpinit() ✓
- e. At any point in time, after main(), the kernel may be parallelly executing on any of the processors. ✓

The correct answers are: xv6 supports only SMP, xv6 supports a variable number of processors (upto 8) and adjusts it's data structures according to number of processors, Each processor on xv6 starts code execution in mpenter() when first processor sets "started" to 1., In xv6 on x86, the first processor configures other processors in mpinit(), At any point in time, after main(), the kernel may be parallelly executing on any of the processors.

Question 19

Partially correct

Mark 0.67 out of 1.00

Which of the following instructions should be privileged?

- a. Issue a trap instruction. ✓
- b. Access I/O device. ✓
- c. Turn off interrupts. ✓
- d. Set value of timer. ✓
- e. Modify entries in device-status table. ✓
- f. Switch from user to kernel mode.
- g. Read the clock.

The correct answers are: Set value of timer., Access I/O device., Issue a trap instruction., Switch from user to kernel mode., Modify entries in device-status table., Turn off interrupts.

Question 20

Correct

Mark 1.00 out of 1.00

Write the possible contents of the file /tmp/xyz after this program.

In the answer if you want to mention any non-text character, then write \0 For example abc\0\0 means abc followed by any two non-text characters

```
int main(int argc, char *argv[]) {  
    int fd1, fd2, n, i;  
    char buf[128];  
  
    fd1 = open("/tmp/xyz", O_WRONLY | O_CREAT, S_IRUSR|S_IWUSR);  
    write(fd1, "hello", 5);  
    fd2 = open("/tmp/xyz", O_WRONLY, S_IRUSR|S_IWUSR);  
    write(fd2, "bye", 3);  
    close(fd1);  
    close(fd2);  
    return 0;  
}
```

Answer: byelo ✓

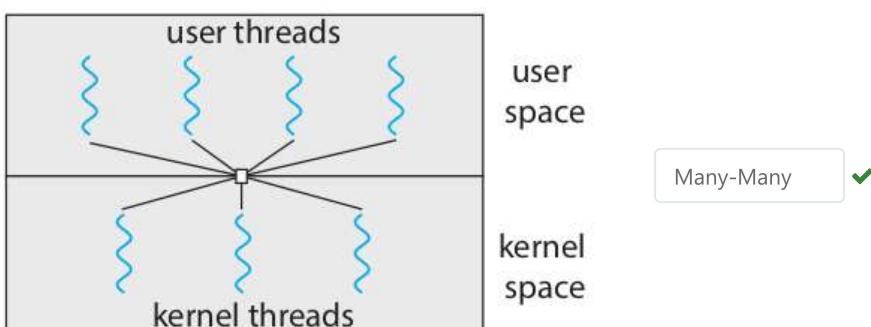
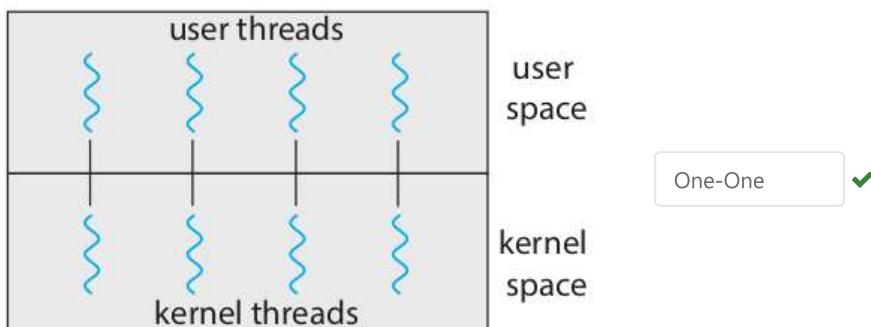
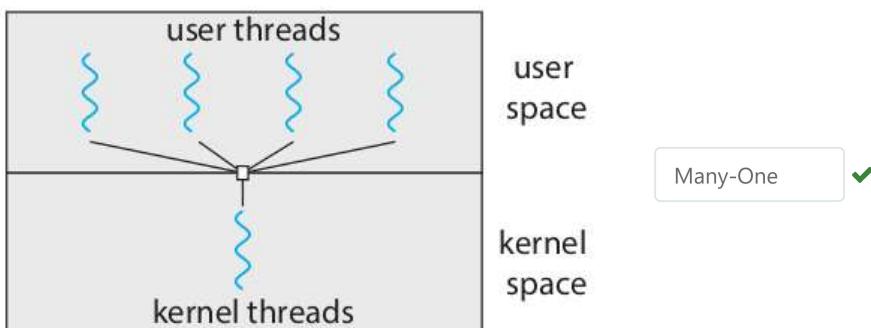
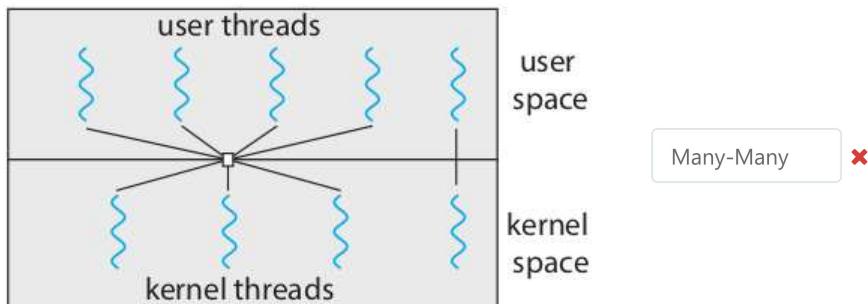
The correct answer is: byelo

Question 21

Partially correct

Mark 0.75 out of 1.00

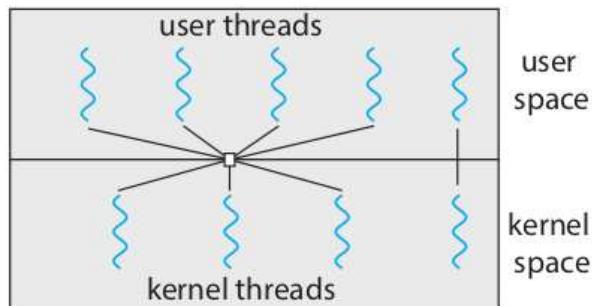
Match the diagram with the threading model



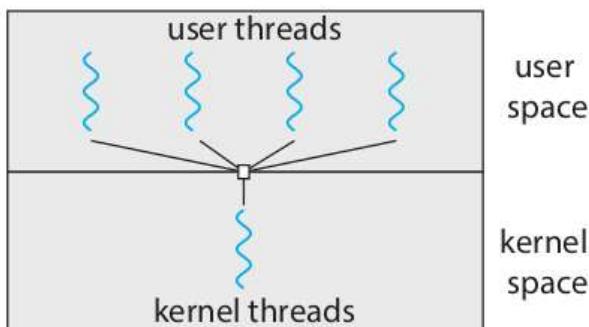
Your answer is partially correct.

You have correctly selected 3.

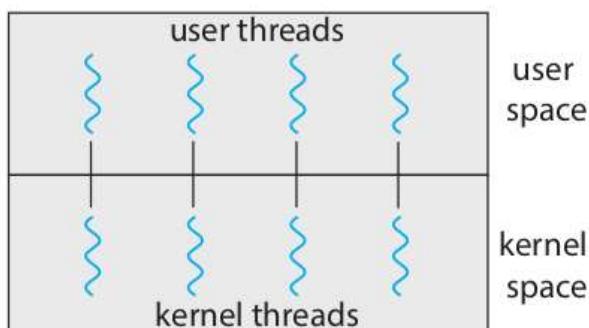
The correct answer is:



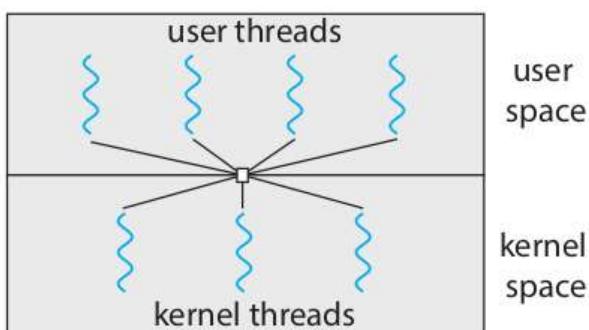
→ Two-Level,



→ Many-One,



→ One-One,



→ Many-Many

Question 22

Correct

Mark 1.00 out of 1.00

Which one of the following is not a scheduling algorithm

- a. Priority
- b. paging
- c. Multilevel Feedback Queue
- d. FCFS



The correct answer is: paging

Question 23

Partially correct

Mark 1.86 out of 2.00

Given below is an incomplete code for reader-writer lock with preference for readers. Fill in the blanks to complete the code.

Note1:

In your answer, if an expression is to be written, then please separate all tokens by exactly one space in between. For example

i = i + 1 is correct while

i=i+1 is not correct or

i= i +1 is also not correct.

Note2:

Correct: a->b++ (no spaces here!)

Any other notation is incorrect.

Note3: The code of downgrade() and upgrade() has proportional to 4/7 marks.

Code of rwlock:

```
typedef struct rwlock {
```

```
    int nActive;
```

```
    int nPendingReads;
```

```
    int nPendingWrites;
```

```
    spinlock_t
```

sl

✓ ;

```
    condition canRead;
```

```
    condition canWrite;
```

```
}rwlock;
```

```
void lockShared(rwlock *r) {
```

```
    spin_lock(&r->sl);
```

r->nPendingReads++

✓ ;

```
    while(r->nActive < 0)
```

```
        wait(
```

&r->canRead

✓ , &r->sl);

r->nActive++;

✗ ;

```
    r->nPendingReads--;
```

```
    spin_unlock(&r->sl);
```

```
}
```

```
void unlockShared(rwlock *r) {
```

```
    spin_lock(&r->sl);
```

r->nActive--

✓ ;

```
    if(r->nActive == 0) {
```

```
        spin_unlock(&r->sl);
```

```
        do_signal(
```

```

&r->canWrite

✓ );
} else
    spin_unlock(&r->sl);
}

void lockExclusive(rwlock *r) {
    spin_lock(&r->sl);
    r->nPendingWrites++;
    while(r->nActive || r->nPendingReads)
        wait(
            &r->canWrite,
            &r->sl
        );
    }

✓ );
r->nPendingWrites--;

```

r->nActive = -1

```

✓ ;
spin_unlock(&r->sl);
}

void unlockExclusive(rwlock *r) {
    boolean_t wakeReaders;
    int i;
    spin_lock(&r->sl);

```

r->nActive = 0

```

✓ ;
if(r->nPendingReads != 0) {
    for(i = 0; i <
        r->nPendingReads
        ; i++)
        do_signal(&r->canRead);
}
else
    do_signal(
        &r->canWrite
    );

```

```

✓ );
spin_unlock(&r->sl);
}

```

```
void downgrade(rwlock *r) {
    boolean_t wakeReaders;
    int i;
    spin_lock(&r->sl);
    r->nActive =

```

1

```

✓ ;
if(r->nPendingReads != 0) {
    for(i = 0; i <
        r->nPendingReads
        ; i++)
        do_signal(

```

```
&r->canRead

✓ );
spin_unlock(&r->sl);
}

void upgrade(rwlock *r) {
    spin_lock(&r->sl);
    if(r->nActive == 1) {
        r->nActive =
            -1
    }
    ;
} else {
    r->nPendingWrites++;
    r->nActive--;
    while(r->nActive != 0)
        )
    wait(
        &r->canWrite
    ,
        , &r->sl);

    r->nPendingWrites--;
    ;
    r->nActive =
        -1
    ;
}
    }
    spin_unlock(&r->sl);
}
```

Question 24

Partially correct

Mark 0.17 out of 1.00

Select the correct statements about hard and soft links

Select one or more:

- a. Deleting a soft link deletes both the link and the actual file ✗
- b. Deleting a hard link always deletes the file ✗
- c. Soft link shares the inode of actual file ✗
- d. Hard links can span across partitions while soft links can't ✗
- e. Deleting a soft link deletes the link, not the actual file ✗
- f. Hard links share the inode ✓
- g. Soft links increase the link count of the actual file inode ✗
- h. Deleting a hard link deletes the file, only if link count was 1 ✓
- i. Deleting a soft link deletes only the actual file ✗
- j. Hard links enforce separation of filename from it's metadata in on-disk data structures. ✓
- k. Hard links increase the link count of the actual file inode ✗
- l. Soft links can span across partitions while hard links can't ✓

Your answer is partially correct.

You have selected too many options.

The correct answers are: Soft links can span across partitions while hard links can't, Hard links increase the link count of the actual file inode, Deleting a soft link deletes the link, not the actual file, Deleting a hard link deletes the file, only if link count was 1, Hard links share the inode, Hard links enforce separation of filename from it's metadata in on-disk data structures.

Question 25

Correct

Mark 1.00 out of 1.00

which of the following scheduling algorithms discriminate either in favor of or against short processes, from the perspective of minimizing waiting time.

For	Against	
<input type="radio"/> ✗	<input checked="" type="radio"/> ✗	FCFS
<input checked="" type="radio"/> ✗	<input type="radio"/> ✗	Multilevel feedback queues
<input checked="" type="radio"/> ✗	<input type="radio"/> ✗	Round Robin

FCFS: Against

Multilevel feedback queues: For

Round Robin: For

Question 26

Correct

Mark 1.00 out of 1.00

Select T/F for the disk block allocation scheme related statements

True	False	
<input checked="" type="radio"/>	<input type="radio"/> X	Continuous allocation allows to fetch any block with just one seek
<input checked="" type="radio"/>	<input type="radio"/> X	Continuous allocation leads to faster file access
<input checked="" type="radio"/>	<input type="radio"/> X	The unix inode is based on indexed allocation
<input checked="" type="radio"/>	<input type="radio"/> X	FAT uses linked allocation
<input checked="" type="radio"/>	<input type="radio"/> X	Linked allocation does away with file size limitation to a large extent
<input checked="" type="radio"/>	<input type="radio"/> X	NVM storage devices are pushing for search for new block allocation schemes
<input checked="" type="radio"/>	<input type="radio"/> X	Continuous allocation may involve a costly search for free space
<input checked="" type="radio"/>	<input type="radio"/> X	Maximum file size limit is determined by disk block allocation scheme, as one of the factors.

Continuous allocation allows to fetch any block with just one seek: True

Continuous allocation leads to faster file access: True

The unix inode is based on indexed allocation: True

FAT uses linked allocation: True

Linked allocation does away with file size limitation to a large extent: True

NVM storage devices are pushing for search for new block allocation schemes: True

Continuous allocation may involve a costly search for free space: True

Maximum file size limit is determined by disk block allocation scheme, as one of the factors.: True

Question 27

Partially correct

Mark 0.50 out of 1.00

Select which of the following data structures may need an update, in ext2 file system, when 5 bytes get removed from the end of an existing file

Yes	No	
<input checked="" type="radio"/>	<input checked="" type="radio"/>	Inode of the file
<input checked="" type="radio"/>	<input checked="" type="radio"/>	The inode of the parent directory
<input checked="" type="radio"/>	<input checked="" type="radio"/>	The inode bitmap
<input checked="" type="radio"/>	<input checked="" type="radio"/>	A free block on the file-system
<input checked="" type="radio"/>	<input checked="" type="radio"/>	One of the block bitmaps
<input checked="" type="radio"/>	<input checked="" type="radio"/>	The boot sector
<input checked="" type="radio"/>	<input checked="" type="radio"/>	The superblock
<input checked="" type="radio"/>	<input checked="" type="radio"/>	The group descriptor

Inode of the file: Yes

The inode of the parent directory: No

The inode bitmap: No

A free block on the file-system: No

One of the block bitmaps: Yes

The boot sector: No

The superblock: Yes

The group descriptor: Yes

Question 28

Partially correct

Mark 0.67 out of 1.00

Consider this program.

Some statements are identified using the // comment at the end.

Assume that = is an atomic operation.

```
#include <stdio.h>
#include <pthread.h>
long c = 0, c1 = 0, c2 = 0, run = 1;
void *thread1(void *arg) {
    while(run == 1) { //E
        c = c1; //A
        c1++; //B
    }
}
void *thread2(void *arg) {
    while(run == 1) { //F
        c = c2; //C
        c2++; //D
    }
}
int main() {
    pthread_t th1, th2;
    pthread_create(&th1, NULL, thread1, NULL);
    pthread_create(&th2, NULL, thread2, NULL);
    sleep(2);
    run = 0;
    printf(stdout, "c = %ld c1+c2 = %ld c1 = %ld c2 = %ld \n", c, c1+c2, c1, c2);
    fflush(stdout);
}
```

Which statements are part of the critical Section?

Yes	No	
<input checked="" type="radio"/> <input checked="" type="checkbox"/>	<input type="radio"/> <input checked="" type="checkbox"/>	D
<input checked="" type="radio"/> <input checked="" type="checkbox"/>	<input checked="" type="radio"/> <input type="checkbox"/>	C
<input type="radio"/> <input checked="" type="checkbox"/>	<input checked="" type="radio"/> <input type="checkbox"/>	F
<input checked="" type="radio"/> <input checked="" type="checkbox"/>	<input checked="" type="radio"/> <input type="checkbox"/>	A
<input checked="" type="radio"/> <input type="checkbox"/>	<input type="radio"/> <input checked="" type="checkbox"/>	B
<input type="radio"/> <input checked="" type="checkbox"/>	<input checked="" type="radio"/> <input type="checkbox"/>	E

As per Remzi's book: "A critical section is a piece of code that accesses a shared variable (or more generally, a shared resource and must not be concurrently executed by more than one thread."

As per Galvin's Book: a critical section, in which the process may be accessing — and updating — data that is shared with at least one other process. The important feature of the system is that when one process is executing in its critical section, no other process is allowed to execute in its critical section.

Since A and C refer to a variable that is shared, it's critical section. Here the hardware guarantees (as = is atomic) that the critical section is accessed by only one thread at a time.

- D: No
- C: Yes
- F: No
- A: Yes
- B: No
- E: No

Question 29

Partially correct

Mark 0.25 out of 0.50

Select all the correct statements related to implementation of a Hypervisor like VirtualBox

- a. The HOST OS determines whether a process (in Guest) or the guest OS itself was doing privileged instruction depending on the privilege level.
- b. When an application runs a privileged instruction it traps into the actual hardware
- c. Typically they run using CPU privilege level 1 or 2 (lower than kernel's but higher than applications) ✓
- d. All Traps in hardware are handled by HOST OS, but Host OS may hand it over to Guest OS if trap was from within guest OS ✓

The correct answers are: Typically they run using CPU privilege level 1 or 2 (lower than kernel's but higher than applications), When an application runs a privileged instruction it traps into the actual hardware, All Traps in hardware are handled by HOST OS, but Host OS may hand it over to Guest OS if trap was from within guest OS, The HOST OS determines whether a process (in Guest) or the guest OS itself was doing privileged instruction depending on the privilege level.

Question 30

Partially correct

Mark 0.75 out of 1.00

For Virtual File System to work, which of the following changes are required to be done to an existing OS code (e.g. xv6)?

- a. The file system specific function pointers, for file system system-calls, need to be setup in the generic inode during lookup.
- b. The filesystem related system calls (e.g. read, write) need to invoke the file system specific functions (e.g. ext2_read, ext2_write, ntfs_read, ntfs_write) using function pointers. ✓
- c. Each file-system writer needs to provide the set of function pointers for VFS, and these function pointers need to be setup in generic inode of "/" of that file system during mount() ✓
- d. Each open() needs to copy the function pointers from the inode of the parent directory into the inode of the child (if not already done), unless it's traversing a mount point. (This may be done as part of lookup() which is called by open()) ✓
- e. The generic inode needs to have a field representing if this inode is a mount point and also to refer/point to the root of the mounted file system's inode.
- f. The lookup() operation needs to check if it's crossing a mount point and call FS specific operations to read inodes/directories ✓
- g. A mount() system call should be provided to mount a partition onto some directory in existing namespace rooted at "/" ✓
- h. The operating system in-memory inode needs to be a generic-inode representing "inode" like data structure across multiple file systems. ✓

The correct answers are: A mount() system call should be provided to mount a partition onto some directory in existing namespace rooted at "/", The filesystem related system calls (e.g. read, write) need to invoke the file system specific functions (e.g. ext2_read, ext2_write, ntfs_read, ntfs_write) using function pointers., The file system specific function pointers, for file system system-calls, need to be setup in the generic inode during lookup., The operating system in-memory inode needs to be a generic-inode representing "inode" like data structure across multiple file systems., The generic inode needs to have a field representing if this inode is a mount point and also to refer/point to the root of the mounted file system's inode., The lookup() operation needs to check if it's crossing a mount point and call FS specific operations to read inodes/directories, Each file-system writer needs to provide the set of function pointers for VFS, and these function pointers need to be setup in generic inode of "/" of that file system during mount(), Each open() needs to copy the function pointers from the inode of the parent directory into the inode of the child (if not already done), unless it's traversing a mount point. (This may be done as part of lookup() which is called by open())

Question 31

Partially correct

Mark 0.29 out of 1.00

Mark the statements as True/False w.r.t. the basic concepts of memory management.

True	False	
<input checked="" type="radio"/>	<input checked="" type="radio"/>	The compiler generates address references for code/data/stack/heap in the executable file, depending on the MM architecture provided by CPU and kernel.
<input checked="" type="radio"/>	<input checked="" type="radio"/>	The compiler interacts with the kernel continuously while compiling a program and obtains the correct set of memory addresses for code/stack/heap/data and then generates the machine code file.
<input checked="" type="radio"/>	<input checked="" type="radio"/>	The kernel ensures that the MMU is setup before scheduling a process and then the CPU/MMU ensures that the address translation takes place.
<input checked="" type="radio"/>	<input checked="" type="radio"/>	When a process is executing, each virtual address is converted into physical address by the CPU hardware directly.
<input checked="" type="radio"/>	<input checked="" type="radio"/>	When a process is executing, each virtual address is converted into physical address by the kernel directly.
<input checked="" type="radio"/>	<input checked="" type="radio"/>	The kernel refers to the page table for converting each virtual address to physical address.
<input checked="" type="radio"/>	<input checked="" type="radio"/>	The compiler generates the address references for code/data/stack/heap in the executable file as per the memory management schema chosen by the compiler itself, and then the kernel ensures that program is executed with this schema.

The compiler generates address references for code/data/stack/heap in the executable file, depending on the MM architecture provided by CPU and kernel.: True

The compiler interacts with the kernel continuously while compiling a program and obtains the correct set of memory addresses for code/stack/heap/data and then generates the machine code file.: False

The kernel ensures that the MMU is setup before scheduling a process and then the CPU/MMU ensures that the address translation takes place.: True

When a process is executing, each virtual address is converted into physical address by the CPU hardware directly.: True

When a process is executing, each virtual address is converted into physical address by the kernel directly.: False

The kernel refers to the page table for converting each virtual address to physical address.: False

The compiler generates the address references for code/data/stack/heap in the executable file as per the memory management schema chosen by the compiler itself, and then the kernel ensures that program is executed with this schema.: False

Question 32

Correct

Mark 1.00 out of 1.00

Consider a computer system with a 32-bit logical address and 8- KB page size. The system supports up to 1 GB of physical memory. How many entries are there in each of the following?

- a. A conventional, single-level page table (write a decimal number):

524288



- b. An inverted page table (number of entries only, write a decimal number):

131072

**Question 33**

Correct

Mark 1.00 out of 1.00

Mark the statements as True/False with respect to Mobile systems and swapping.

True	False	
<input checked="" type="radio"/>	<input type="radio"/>	Poor throughput between main memory and flash memory is one reason for restriction on use of swap on mobile devices.
<input checked="" type="radio"/>	<input type="radio"/>	Size of flash memory is one reason for restriction on use of swap on mobile devices.
<input checked="" type="radio"/>	<input type="radio"/>	Mobile systems generally do not support swapping
<input checked="" type="radio"/>	<input type="radio"/>	Limited number of write operations that flash memory can tolerate, is one reason for limitations of swapping on mobile devices.

Poor throughput between main memory and flash memory is one reason for restriction on use of swap on mobile devices.: True

Size of flash memory is one reason for restriction on use of swap on mobile devices.: True

Mobile systems generally do not support swapping: True

Limited number of write operations that flash memory can tolerate, is one reason for limitations of swapping on mobile devices.: True

Question 34

Partially correct

Mark 1.38 out of 3.00

Suppose it is required to add the chown() (without notion of a group, just the notion of owner and others) system call to xv6. Select the changes, from the options given below, which are an absolute must to effect the addition of this system call.

Changes w.r.t. other system calls should be selected if those system calls are necessary for the implementation of the chown() system call.

Must	Not-Must	
<input checked="" type="radio"/> ✘	<input type="checkbox"/> ✓	Add a setuid(), seteuid() system call, callable only by the user with ID or EUID equal to "0" to set the new user id of the process
<input checked="" type="radio"/> ✘	<input type="checkbox"/> ✓	Create an application program su.c which itself is a SUID program, and calls setuid() and seteuid() with specified user-ID and then does exec() of a shell.
<input checked="" type="radio"/> ✘	<input checked="" type="checkbox"/>	Modify exec() to check SUID bit on the executable file and set EUID of the process to UID of the executable
<input checked="" type="checkbox"/>	<input checked="" type="radio"/> ✘	Add a mode field representing file permissions, inside dinode
<input checked="" type="radio"/> ✘	<input type="checkbox"/> ✓	Add a UID field in the struct proc representing the USER-ID of the user who started this process
<input checked="" type="radio"/> ✘	<input type="checkbox"/> ✓	Mandatorily create a file like "passwd" which maps user-names to user-IDs and modify other utilities (like "ls") to show user-name instead of user-ID for the files.
<input checked="" type="radio"/> ✘	<input checked="" type="checkbox"/>	Add the username field to the on disk inode, that is dinode.
<input checked="" type="radio"/> ✘	<input type="checkbox"/> ✓	Inherit the UID and EUID of the parent in fork()
<input checked="" type="checkbox"/>	<input checked="" type="radio"/> ✘	Add a uid field representing the owner, inside dinode
<input checked="" type="radio"/> ✘	<input type="checkbox"/> ✓	Add few extra files belonging to different users, using mkfs.c
<input checked="" type="checkbox"/>	<input checked="" type="radio"/> ✘	Modify mkfs.c to add owner and permissions to each file created
<input checked="" type="checkbox"/>	<input checked="" type="radio"/> ✘	Add the chown() system call which checks if the user with id "0" is calling this system call and then change the ownership of the on-disk and in-memory inode.

Not-Must	Not-Must
<input checked="" type="radio"/> X	<input type="checkbox"/> ✓

Add a EUID field in the struct proc representing Effective user ID

X

Add a setuid(), seteuid() system call, callable only by the user with ID or EUID equal to "0" to set the new user id of the process: Not-Must
Create an application program su.c which itself is a SUID program, and calls setuid() and seteuid() with specified user-ID and then does exec() of a shell.: Not-Must

Modify exec() to check SUID bit on the executable file and set EUID of the process to UID of the executable: Not-Must

Add a mode field representing file permissions, inside dinode: Must

Add a UID field in the struct proc representing the USER-ID of the user who started this process: Not-Must

Mandatorily create a file like "passwd" which maps user-names to user-IDs and modify other utilities (like "ls") to show user-name instead of user-ID for the files.: Not-Must

Add the username field to the on disk inode, that is dinode.: Not-Must

Inherit the UID and EUID of the parent in fork(): Not-Must

Add a uid field representing the owner, inside dinode: Must

Add few extra files belonging to different users, using mkfs.c: Not-Must

Modify mkfs.c to add owner and permissions to each file created: Must

Add the chown() system call which checks if the user with id "0" is calling this system call and then change the ownership of the on-disk and in-memory inode.: Must

Add a EUID field in the struct proc representing Effective user ID: Not-Must

Question 35

Partially correct

Mark 0.86 out of 1.00

Mark statements as T/F

All statements are in the context of preventing deadlocks.

True	False	
<input checked="" type="radio"/>	<input type="radio"/>	Mutual exclusion is a necessary condition for deadlock because it brings in locks on which deadlock happens
<input checked="" type="radio"/>	<input type="radio"/>	Circular wait is avoided by enforcing a lock ordering
<input checked="" type="radio"/>	<input type="radio"/>	The lock ordering to be followed to avoid circular wait is a protocol to be followed by programmers.
<input checked="" type="radio"/>	<input type="radio"/>	If a resource allocation graph contains a cycle then there is a possibility of a deadlock
<input type="radio"/>	<input checked="" type="radio"/>	If a resource allocation graph contains a cycle then there is a guarantee of a deadlock
<input type="radio"/>	<input checked="" type="radio"/>	The lock ordering to be followed to avoid circular wait is a code in OS that checks for compliance with decided order
<input checked="" type="radio"/>	<input type="radio"/>	Deadlock is not possible if any of these conditions is not met: Mutual exclusion, hold and wait, no pre-emption, circular wait.

Mutual exclusion is a necessary condition for deadlock because it brings in locks on which deadlock happens: True
Circular wait is avoided by enforcing a lock ordering: True

The lock ordering to be followed to avoid circular wait is a protocol to be followed by programmers.: True

If a resource allocation graph contains a cycle then there is a possibility of a deadlock: True

If a resource allocation graph contains a cycle then there is a guarantee of a deadlock: False

The lock ordering to be followed to avoid circular wait is a code in OS that checks for compliance with decided order: False

Deadlock is not possible if any of these conditions is not met: Mutual exclusion, hold and wait, no pre-emption, circular wait.: True

[◀ Course Exit Feedback](#)

Jump to...

Started on Wednesday, 19 April 2023, 6:30 PM

State Finished

Completed on Wednesday, 19 April 2023, 8:52 PM

Time taken 2 hours 21 mins

Overdue 21 mins 46 secs

Grade 23.26 out of 30.00 (77.54%)

Question 1

Partially correct

Mark 0.88 out of 1.00

Select all correct statements about file system recovery (without journaling) programs e.g. fsck

Select one or more:

- a. A recovery program, most typically, builds the file system data structure and checks for inconsistencies
- b. Recovery programs are needed only if the file system has a delayed-write policy. ✓
- c. They may take very long time to execute ✓
- d. It is possible to lose data as part of recovery ✓
- e. Even with a write-through policy, it is possible to need a recovery program. ✓
- f. They can make changes to the on-disk file system ✓
- g. Recovery is possible due to redundancy in file system data structures ✓
- h. They are used to recover deleted files
- i. Recovery programs recalculate most of the metadata summaries (e.g. free inode count) ✓

Your answer is partially correct.

You have correctly selected 7.

The correct answers are: Recovery is possible due to redundancy in file system data structures, A recovery program, most typically, builds the file system data structure and checks for inconsistencies, It is possible to lose data as part of recovery, They may take very long time to execute, They can make changes to the on-disk file system, Recovery programs recalculate most of the metadata summaries (e.g. free inode count), Recovery programs are needed only if the file system has a delayed-write policy., Even with a write-through policy, it is possible to need a recovery program.

Question 2

Partially correct

Mark 1.75 out of 2.00

Match the snippets of xv6 code with the core functionality they achieve, or problems they avoid.

"..." means some code.

```
void  
acquire(struct spinlock *lk)  
{  
...  
__sync_synchronize();
```

Tell compiler not to reorder memory access beyond this line



```
void  
yield(void)  
{  
...  
release(&ptable.lock);  
}
```

Release the lock held by some another process



```
void  
acquire(struct spinlock *lk)  
{  
...  
getcallerpcs(&lk, lk->pcs);
```

Traverse ebp chain to get sequence of instructions followed in functions calls



```
void  
acquire(struct spinlock *lk)  
{  
pushcli();
```

Disable interrupts to avoid deadlocks



```
void  
panic(char *s)  
{  
...  
panicked = 1;
```

Ensure that no printing happens on other processors



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

If you don't do this, a process may be running on two processors parallely



```

static inline uint
xchg(volatile uint *addr, uint newval)
{
    uint result;

    // The + in "+m" denotes a read-modify-write
    // operand.
    asm volatile("lock; xchgl %0, %1" :
        "+m" (*addr), "=a" (result) :
        "1" (newval) :
        "cc");
    return result;
}

struct proc*
myproc(void) {
...
pushcli();
c = mycpu();
p = c->proc;
popcli();
...
}

```

Atomic compare and swap instruction (to be expanded inline into code)



Disable interrupts to avoid another process's pointer being returned



Your answer is partially correct.

You have correctly selected 7.

The correct answer is: `void
acquire(struct spinlock *lk)
{
...
__sync_synchronize();`

→ Tell compiler not to reorder memory access beyond this line, `void`

```

yield(void)
{
...
release(&ptable.lock);
}
```

→ Release the lock held by some another process, `void`

```

acquire(struct spinlock *lk)
{
...
getcallerpcs(&lk, lk->pcs);
```

→ Traverse ebp chain to get sequence of instructions followed in functions calls, `void`

```

acquire(struct spinlock *lk)
{
    pushcli();
    → Disable interrupts to avoid deadlocks, void
    panic(char *s)
{
...
panicked = 1; → Ensure that no printing happens on other processors, void
```

```

sleep(void *chan, struct spinlock *lk)
{
    ...
    if(lk != &ptable.lock){
        acquire(&ptable.lock);
        release(lk);
    } → Avoid a self-deadlock, static inline uint
xchg(volatile uint *addr, uint newval)
{
    uint result;

    // The + in "+m" denotes a read-modify-write operand.
    asm volatile("lock; xchgl %0, %1" :
        "+m" (*addr), "=a" (result) :
        "1" (newval) :
        "cc");
    return result;
} → Atomic compare and swap instruction (to be expanded inline into code), struct proc*
myproc(void) {
    ...
    pushcli();
    c = mycpu();
    p = c->proc;
    popcli();
    ...
}

```

→ Disable interrupts to avoid another process's pointer being returned

Question 3

Partially correct

Mark 0.80 out of 1.00

Select all the correct statements w.r.t user and kernel threads

Select one or more:

- a. all three models, that is many-one, one-one, many-many , require a user level thread library✓
- b. many-one model gives no speedup on multicore processors✓
- c. one-one model can be implemented even if there are no kernel threads
- d. many-one model can be implemented even if there are no kernel threads✓
- e. one-one model increases kernel's scheduling load
- f. A process may not block in many-one model, if a thread makes a blocking system call
- g. A process blocks in many-one model even if a single thread makes a blocking system call✓

Your answer is partially correct.

You have correctly selected 4.

The correct answers are: many-one model can be implemented even if there are no kernel threads, all three models, that is many-one, one-one, many-many , require a user level thread library, one-one model increases kernel's scheduling load, many-one model gives no speedup on multicore processors, A process blocks in many-one model even if a single thread makes a blocking system call

Question 4

Correct

Mark 2.00 out of 2.00

Select all the correct statements about synchronization primitives.

Select one or more:

- a. Blocking means one process passing over control to another process
- b. Semaphores are always a good substitute for spinlocks
- c. Spinlocks consume CPU time ✓
- d. Mutexes can be implemented using blocking and wakeup ✓
- e. All synchronization primitives are implemented essentially with some hardware assistance. ✓
- f. Mutexes can be implemented without any hardware assistance
- g. Spinlocks are good for multiprocessor scenarios, for small critical sections ✓
- h. Semaphores can be used for synchronization scenarios like ordered execution ✓
- i. Mutexes can be implemented using spinlock ✓
- j. Blocking means moving the process to a wait queue and calling scheduler ✓
- k. Thread that is going to block should not be holding any spinlock ✓
- l. Blocking means moving the process to a wait queue and spinning

Your answer is correct.

The correct answers are: Spinlocks are good for multiprocessor scenarios, for small critical sections, Spinlocks consume CPU time, Semaphores can be used for synchronization scenarios like ordered execution, Mutexes can be implemented using spinlock, Mutexes can be implemented using blocking and wakeup, Thread that is going to block should not be holding any spinlock, Blocking means moving the process to a wait queue and calling scheduler, All synchronization primitives are implemented essentially with some hardware assistance.

Question 5

Correct

Mark 1.00 out of 1.00

Match the snippets of xv6 code with the core functionality they achieve, or problems they avoid.

"..." means some code.

```
void
acquire(struct spinlock *lk)
{
...
getcallerpcs(&lk, lk->pcs);
```

Traverse ebp chain to get sequence of instructions followed in functions calls

✓

```
void
yield(void)
{
...
release(&ptable.lock);
```

Release the lock held by some another process

✓

```
void
panic(char *s)
{
...
panicked = 1;
```

Ensure that no printing happens on other processors

✓

Your answer is correct.

The correct answer is:

```
void
acquire(struct spinlock *lk)
{
...
getcallerpcs(&lk, lk->pcs); → Traverse ebp chain to get sequence of instructions followed in functions calls, void
yield(void)
{
...
release(&ptable.lock);
} → Release the lock held by some another process, void
panic(char *s)
{
...
panicked = 1; → Ensure that no printing happens on other processors
```

Question 6

Correct

Mark 1.00 out of 1.00

Given that a kernel has 1000 KB of total memory, and holes of sizes (in that order) 300 KB, 200 KB, 100 KB, 250 KB. For each of the requests on the left side, match it with the chunk chosen using the specified algorithm.

Consider each request as first request.

50 KB, worst fit	300 KB	✓
200 KB, first fit	300 KB	✓
150 KB, first fit	300 KB	✓
220 KB, best fit	250 KB	✓
100 KB, worst fit	300 KB	✓
150 KB, best fit	200 KB	✓

The correct answer is: 50 KB, worst fit → 300 KB, 200 KB, first fit → 300 KB, 150 KB, first fit → 300 KB, 220 KB, best fit → 250 KB, 100 KB, worst fit → 300 KB, 150 KB, best fit → 200 KB

Question 7

Correct

Mark 1.00 out of 1.00

Mark the statements as True or False, w.r.t. passing of arguments to system calls in xv6 code.

True	False	
<input checked="" type="radio"/>	<input type="radio"/>	The functions like argint(), argstr() make the system call arguments available in the kernel.
<input checked="" type="radio"/>	<input type="radio"/>	The arguments are accessed in the kernel code using esp on the trapframe.
<input type="radio"/>	<input checked="" type="radio"/>	String arguments are first copied to trapframe and then from trapframe to kernel's other variables.
<input checked="" type="radio"/>	<input type="radio"/>	Integer arguments are copied from user memory to kernel memory using argint()
<input type="radio"/>	<input checked="" type="radio"/>	Integer arguments are stored in eax, ebx, ecx, etc. registers
<input checked="" type="radio"/>	<input type="radio"/>	The arguments to system call originally reside on process stack.
<input checked="" type="radio"/>	<input type="radio"/>	String arguments are NOT copied in kernel memory, but just pointed to by a kernel memory pointer
<input type="radio"/>	<input checked="" type="radio"/>	The arguments to system call are copied to kernel stack in trapasm.S

The functions like argint(), argstr() make the system call arguments available in the kernel.: True

The arguments are accessed in the kernel code using esp on the trapframe.: True

String arguments are first copied to trapframe and then from trapframe to kernel's other variables.: False

Integer arguments are copied from user memory to kernel memory using argint(): True

Integer arguments are stored in eax, ebx, ecx, etc. registers: False

The arguments to system call originally reside on process stack.: True

String arguments are NOT copied in kernel memory, but just pointed to by a kernel memory pointer: True

The arguments to system call are copied to kernel stack in trapasm.S: False

Question 8

Correct

Mark 1.00 out of 1.00

Note: for this question you get full marks if you select all and only correct options, you get ZERO if at least one option is wrong or not selected.

Select all the correct statements about log structured file systems.

- a. file system recovery recovers all the lost data
- b. file system recovery may end up losing data✓
- c. log may be kept on same block device or another block device✓
- d. even if file systems followed immediate writes (i.e. non-delayed writes), it could still require recovery✓
- e. a transaction is said to be committed when all operations are written to file system

Your answer is correct.

The correct answers are: file system recovery may end up losing data, log may be kept on same block device or another block device, even if file systems followed immediate writes (i.e. non-delayed writes), it could still require recovery

Question 9

Complete

Mark 1.50 out of 3.00

List down all changes required to xv6 code, in order to add the system call chown().

Every change should be mentioned in terms of either of the following:

- (a) pseudo-code of new function to be added
- (b) prototype of any new function or new system call to be added
- (c) pseudo-code of changes to an existing function, describing lines to be removed, and lines to be added
- (d) **precise** declaration of new data structures to be added in C, or changes to the existing data structure
- (e) Name and a one-line description of new userland functionality to be added
- (f) Changes to Makefile
- (g) Any other change in a maximum of 20 words per change.

This implementation assumes there is multiple user support for xv6

- a) void chown(char* pathname, int owner, int group){
 if(priviledge(currentOwner) > privileged(owner)){
 //change owner in inode of file/folder pointed by pathname
 }
}
- b) sys_calls to check currentOwner() and filePriviledges(char* filepath)
- c) all file related sys_calls like open and read should check for currently logged in user and priviledges for the file
- d)
- e)
struct inode {
 uint dev; // Device number
 uint inum; // Inode number
 int ref; // Reference count
 struct sleeplock lock; // protects everything below here
 int valid; // inode has been read from disk?
 short type; // copy of disk inode
 short major;
 short minor;
 short nlink;
 uint size;
 uint addrs[NDIRECT+1];
 int owner; -----> owner
 int group; -----> group
};
- f) no changes in makefile for adding system a sys_call
- g)

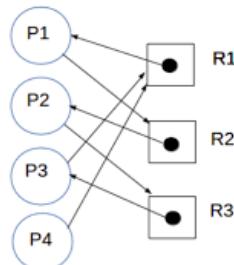
Comment:

Question 10

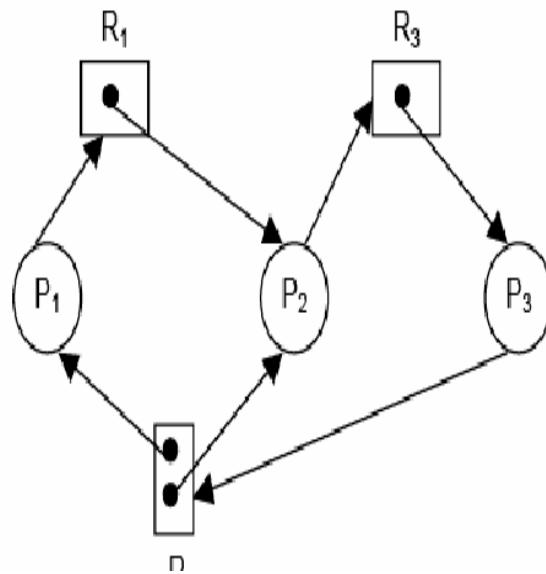
Correct

Mark 1.00 out of 1.00

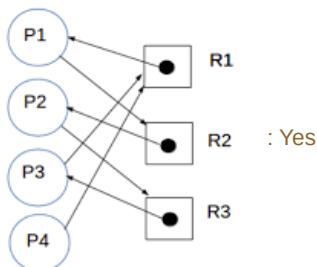
For each of the resource allocation diagram shown,
infer whether the graph contains at least one deadlock or not.

Yes**No** X

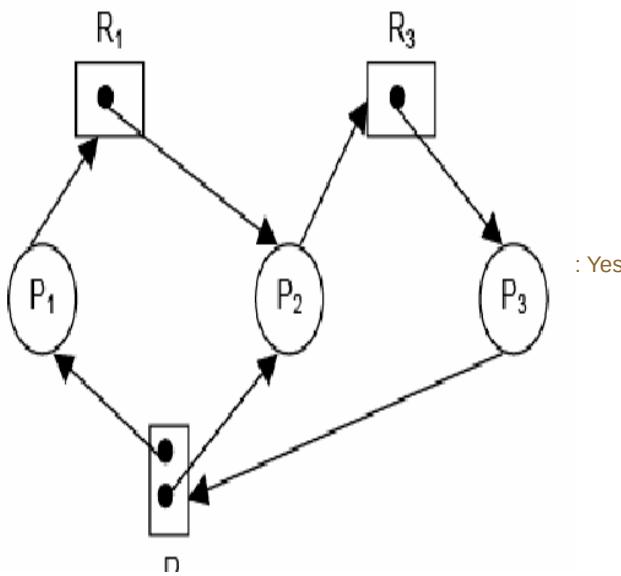
✓

 X

✓



: Yes



: Yes

Question 11

Incorrect

Mark 0.00 out of 1.00

Given that the memory access time is 150 ns, probability of a page fault is 0.8 and page fault handling time is 6 ms,
The effective memory access time in nanoseconds is:

Answer: ×

The correct answer is: 4800030.00

Question 12

Correct

Mark 1.00 out of 1.00

Suppose a kernel uses a buddy allocator. The smallest chunk that can be allocated is of size 32 bytes. One bit is used to track each such chunk, where 1 means allocated and 0 means free. The chunk looks like this as of now:

11010010

Now, there is a request for a chunk of 45 bytes.

After this allocation, the bitmap, indicating the status of the buddy allocator will be

Answer:



The correct answer is: 11011110

Question 13

Partially correct

Mark 0.86 out of 1.00

Select T/F for statements about Volume Managers.

Do pay attention to the use of the words physical partition and physical volume.

True	False	
<input checked="" type="radio"/>	<input type="radio"/> X	The volume manager stores additional metadata on the physical disk partitions
<input checked="" type="radio"/>	<input type="radio"/> X	A logical volume can be extended in size but upto the size of volume group
<input checked="" type="radio"/>	<input type="radio"/> X	A logical volume may span across multiple physical partitions
<input checked="" type="radio"/>	<input type="radio"/> X	A physical partition should be initialized as a physical volume, before it can be used by volume manager.
<input checked="" type="radio"/>	<input type="radio"/> X	A logical volume may span across multiple physical volumes
<input checked="" type="radio"/>	<input type="radio"/> X	The volume manager can create further internal sub-divisions of a physical partition for efficiency or features.
<input checked="" type="radio"/>	<input type="radio"/> X	A volume group consists of multiple physical volumes

The volume manager stores additional metadata on the physical disk partitions: True

A logical volume can be extended in size but upto the size of volume group: True

A logical volume may span across multiple physical partitions: True

A physical partition should be initialized as a physical volume, before it can be used by volume manager.: True

A logical volume may span across multiple physical volumes: True

The volume manager can create further internal sub-divisions of a physical partition for efficiency or features.: True

A volume group consists of multiple physical volumes: True

Question 14

Incorrect

Mark 0.00 out of 1.00

Assuming a 8- KB page size, what is the page numbers for the address 26583 reference in decimal :

(give answer also in decimal)

Answer: 13

X

The correct answer is: 3

Question 15

Correct

Mark 2.00 out of 2.00

For Virtual File System to work, which of the following changes are required to be done to an existing OS code (e.g. xv6)?

- a. Each file-system writer needs to provide the set of function pointers for VFS, and these function pointers need to be setup in generic inode of "/" of that file system during mount() ✓
- b. The generic inode needs to have a field representing if this inode is a mount point and also to refer/point to the root of the mounted file system's inode. ✓
- c. Each open() needs to copy the function pointers from the inode of the parent directory into the inode of the child (if not already done), unless it's traversing a mount point. (This may be done as part of lookup() which is called by open()) ✓
- d. The file system specific function pointers, for file system system-calls, need to be setup in the generic inode during lookup. ✓
- e. The filesystem related system calls (e.g. read, write) need to invoke the file system specific functions (e.g. ext2_read, ext2_write, ntfs_read, ntfs_write) using function pointers. ✓
- f. A mount() system call should be provided to mount a partition onto some directory in existing namespace rooted at "/" ✓
- g. The lookup() operation needs to check if it's crossing a mount point and call FS specific operations to read inodes/directories ✓
- h. The operating system in-memory inode needs to be a generic-inode representing "inode" like data structure across multiple file systems. ✓

The correct answers are: A mount() system call should be provided to mount a partition onto some directory in existing namespace rooted at "/", The filesystem related system calls (e.g. read, write) need to invoke the file system specific functions (e.g. ext2_read, ext2_write, ntfs_read, ntfs_write) using function pointers., The file system specific function pointers, for file system system-calls, need to be setup in the generic inode during lookup., The operating system in-memory inode needs to be a generic-inode representing "inode" like data structure across multiple file systems., The generic inode needs to have a field representing if this inode is a mount point and also to refer/point to the root of the mounted file system's inode., The lookup() operation needs to check if it's crossing a mount point and call FS specific operations to read inodes/directories, Each file-system writer needs to provide the set of function pointers for VFS, and these function pointers need to be setup in generic inode of "/" of that file system during mount(), Each open() needs to copy the function pointers from the inode of the parent directory into the inode of the child (if not already done), unless it's traversing a mount point. (This may be done as part of lookup() which is called by open())

Question 16

Partially correct

Mark 1.43 out of 2.00

Compare paging with demand paging and select the correct statements.

Select one or more:

- a. With paging, it's possible to have user programs bigger than physical memory.
- b. TLB hit ration has zero impact in effective memory access time in demand paging.
- c. Both demand paging and paging support shared memory pages. ✓
- d. With demand paging, it's possible to have user programs bigger than physical memory. ✓
- e. Paging requires NO hardware support in CPU
- f. The meaning of valid-invalid bit in page table is different in paging and demand-paging. ✓
- g. Demand paging always increases effective memory access time.
- h. Paging requires some hardware support in CPU ✓
- i. Calculations of number of bits for page number and offset are same in paging and demand paging.
- j. Demand paging requires additional hardware support, compared to paging. ✓

Your answer is partially correct.

You have correctly selected 5.

The correct answers are: Demand paging requires additional hardware support, compared to paging., Both demand paging and paging support shared memory pages., With demand paging, it's possible to have user programs bigger than physical memory., Demand paging always increases effective memory access time., Paging requires some hardware support in CPU, Calculations of number of bits for page number and offset are same in paging and demand paging., The meaning of valid-invalid bit in page table is different in paging and demand-paging.

Question 17

Partially correct

Mark 0.80 out of 1.00

Mark the statements as True or False, w.r.t. thrashing

True	False	
<input type="radio"/> ✗	<input checked="" type="radio"/> ✗	Processes keep changing their locality of reference, and least number of page faults occur when they are changing the locality.
<input checked="" type="radio"/> ✗	<input type="radio"/> ✗	Thrashing occurs when the total size of all process's locality exceeds total memory size.
<input checked="" type="radio"/> ✗	<input type="radio"/> ✗	Thrashing can be limited if local replacement is used.
<input checked="" type="radio"/> ✗	<input type="radio"/> ✗	During thrashing the CPU is under-utilised as most time is spent in I/O
<input type="radio"/> ✗	<input checked="" type="radio"/> ✗	Thrashing can occur even if entire memory is not in use.
<input checked="" type="radio"/> ✗	<input type="radio"/> ✗	The working set model is an attempt at approximating the locality of a process.
<input checked="" type="radio"/> ✗	<input type="radio"/> ✗	Thrashing is particular to demand paging systems, and does not apply to pure paging systems.
<input type="radio"/> ✗	<input checked="" type="radio"/> ✗	Thrashing occurs because some process is doing lot of disk I/O.
<input type="radio"/> ✗	<input checked="" type="radio"/> ✗	mmap() solves the problem of thrashing.
<input checked="" type="radio"/> ✗	<input type="radio"/> ✗	Processes keep changing their locality of reference, and a high rate of page faults occur when they are changing the locality.

Processes keep changing their locality of reference, and least number of page faults occur when they are changing the locality.: False
Thrashing occurs when the total size of all process's locality exceeds total memory size.: True

Thrashing can be limited if local replacement is used.: True

During thrashing the CPU is under-utilised as most time is spent in I/O: True

Thrashing can occur even if entire memory is not in use.: False

The working set model is an attempt at approximating the locality of a process.: True

Thrashing is particular to demand paging systems, and does not apply to pure paging systems.: True

Thrashing occurs because some process is doing lot of disk I/O.: False

mmap() solves the problem of thrashing.: False

Processes keep changing their locality of reference, and a high rate of page faults occur when they are changing the locality.: True

Question 18

Correct

Mark 1.00 out of 1.00

Match the code with its functionality

S = 5

Wait(S)

Critical Section

Counting semaphore



Signal(S)

S1 = 0; S2 = 0;

P2:

Statement1;

Signal(S2);

P1:

Wait(S2);

Statemetn2;

Signal(S1);

Execution order P2, P1, P3



P3:

Wait(S1);

Statement S3;

S = 0

P1:

Statement1;

Signal(S)

Execution order P1, then P2



P2:

Wait(S)

Statment2;

S = 1

Wait(S)

Critical Section

Binary Semaphore for mutual exclusion



Signal(S);

Your answer is correct.

The correct answer is: S = 5

Wait(S)

Critical Section

Signal(S) → Counting semaphore, S1 = 0; S2 = 0;

P2:

Statement1;

Signal(S2);

P1:

Wait(S2);

Statemetn2;

Signal(S1);

P3:

Wait(S1);

Statement S3; → Execution order P2, P1, P3, S = 0

P1:

Statement1;

Signal(S)

P2:

Wait(S)

Statement2; → Execution order P1, then P2, **S = 1**

Wait(S)

Critical Section

Signal(S); → Binary Semaphore for mutual exclusion

Question 19

Correct

Mark 1.00 out of 1.00

Map the technique with its feature/problem

dynamic linking	small executable file	✓
static loading	wastage of physical memory	✓
static linking	large executable file	✓
dynamic loading	allocate memory only if needed	✓

The correct answer is: dynamic linking → small executable file, static loading → wastage of physical memory, static linking → large executable file, dynamic loading → allocate memory only if needed

Question 20

Partially correct

Mark 0.25 out of 1.00

Select all correct statements about journaling (logging) in file systems like ext3

Select one or more:

- a. A different device driver is always needed to access the journal
- b. Journals are often stored circularly
- c. Most typically a transaction in journal is recorded atomically (full or none) ✓
- d. Journals must be maintained on the same device that hosts the file system
- e. The purpose of journal is to speed up file system recovery
- f. Journal is hosted in the same device that hosts the swap space
- g. the journal contains a summary of all changes made as part of a single transaction

Your answer is partially correct.

You have correctly selected 1.

The correct answers are: The purpose of journal is to speed up file system recovery, the journal contains a summary of all changes made as part of a single transaction, Most typically a transaction in journal is recorded atomically (full or none), Journals are often stored circularly

Question 21

Complete

Mark 1.00 out of 2.00

Write all changes required to xv6 to add a buddy allocator.

Every change should be mentioned in terms of either of the following:

- (a) pseudo-code of new function to be added
- (b) prototype of any new function or new system call to be added
- (c) pseudo-code of changes to an existing function, describing lines to be removed, and lines to be added
- (d) **precise** declaration of new data structures to be added in C, or changes to the existing data structure
- (e) Name and a one-line description of new userland functionality to be added
- (f) Changes to Makefile
- (g) Any other change in a maximum of 20 words per change.

a) buddyAllocate(int requiredSize, string allocatedBitmap, int i, int j){
 string sizeRequired = find the least bigger power of 2 than requiredSize in a form of bitmap
 if(sizeRequired == allocatedBitmap[i:j]) {
 return location in bitmap which match with substring allocatedBitmap[i:j];
 buddyAllocate(requiredSize, 0, sizeof(allocatedBitmap)/2));
 buddyAllocate(requiredSize, sizeof(allocatedBitmap)/2+1, sizeof(allocatedBitmap));

b)
c) filealloc() // remove lines ---->

// for(f = ftable.file; f < ftable.file + NFILE; f++){
// if(f->ref == 0){
// f->ref = 1;
// release(&ftable.lock);
// return f;
// }
// }
and add -----> f = getCache();

d) struct memoryBitmap{
 spinlock sl;
 uint allocatedBitmap;
 int sizeMappedToEachBitmap; // like 32 bytes in one of previous questions

- e)
- f) no changes
- g) nope

Comment:

checked

Question 22

Correct

Mark 1.00 out of 1.00

Calculate the average waiting time using
FCFS scheduling
for the following workload
assuming that they arrive in this order during the first time unit:

Process Burst Time

P1	2
P2	6
P3	2
P4	3

Write only a number in the answer upto two decimal points.

Answer:



P2 waits for 2 units

P3 waits for 2+6 units

P4 waits for 2 + 6 +2 units of time

Total waiting = $2 + 2 + 6 + 2 + 6 + 2 = 20$ units

Average waiting time = $20/4 = 5$

The correct answer is: 5

Question 23

Correct

Mark 1.00 out of 1.00

Match each suggested semaphore implementation (discussed in class)

with the problems that it faces

```
struct semaphore {  
    int val;  
    spinlock lk;  
    list l;  
};  
sem_init(semaphore *s, int initval) {  
    s->val = initval;  
    s->sl = 0;  
}  
block(semaphore *s) {  
    listappend(s->l, current);  
    schedule();  
}  
wait(semaphore *s) {  
    spinlock(&(s->sl));  
    while(s->val <=0) {  
        block(s);  
    }  
    (s->val)--;  
    spinunlock(&(s->sl));  
}
```

blocks holding a spinlock



```
struct semaphore {  
    int val;  
    spinlock lk;  
};  
sem_init(semaphore *s, int initval) {  
    s->val = initval;  
    s->sl = 0;  
}  
wait(semaphore *s) {  
    spinlock(&(s->sl));  
    while(s->val <=0) {  
        spinunlock(&(s->sl));  
        spinlock(&(s->sl));  
    }  
    (s->val)--;  
    spinunlock(&(s->sl));  
}
```

too much spinning, bounded wait not guaranteed



```
struct semaphore {  
    int val;  
    spinlock lk;  
};  
sem_init(semaphore *s, int initval) {  
    s->val = initval;  
    s->sl = 0;  
}  
wait(semaphore *s) {  
    spinlock(&(s->sl));  
    while(s->val <=0)  
    ;  
    (s->val)--;  
    spinunlock(&(s->sl));  
}
```

deadlock



```

struct semaphore {
    int val;
    spinlock lk;
    list l;
};

sem_init(semaphore *s, int initval) {
    s->val = initval;
    s->sl = 0;
}

block(semaphore *s) {
    listappend(s->l, current);
    spinunlock(&(s->sl));
    schedule();
}

wait(semaphore *s) {
    spinlock(&(s->sl));
    while(s->val <=0) {
        block(s);
    }
    (s->val)--;
    spinunlock(&(s->sl));
}

signal(semaphore *s) {
    spinlock(*(s->sl));
    (s->val)++;
    x = dequeue(s->sl) and enqueue(readyq, x);
    spinunlock(*(s->sl));
}

```

not holding lock after unblock ✓

Your answer is correct.

The correct answer is:

```

struct semaphore {
    int val;
    spinlock lk;
    list l;
};

sem_init(semaphore *s, int initval) {
    s->val = initval;
    s->sl = 0;
}

block(semaphore *s) {
    listappend(s->l, current);
    schedule();
}

wait(semaphore *s) {
    spinlock(&(s->sl));
    while(s->val <=0) {
        block(s);
    }
    (s->val)--;
    spinunlock(&(s->sl));
}

```

→ blocks holding a spinlock,

```

struct semaphore {
    int val;
    spinlock lk;
};

sem_init(semaphore *s, int initval) {
    s->val = initval;
    s->sl = 0;
}

wait(semaphore *s) {
    spinlock(&(s->sl));
    while(s->val <=0) {
        spinunlock(&(s->sl));
        spinlock(&(s->sl));
    }
    (s->val)--;
    spinunlock(&(s->sl));
}

```

→ too much spinning, bounded wait not guaranteed,

```

struct semaphore {
    int val;
    spinlock lk;
};

sem_init(semaphore *s, int initval) {
    s->val = initval;
    s->sl = 0;
}

wait(semaphore *s) {
    spinlock(&(s->sl));
    while(s->val <=0)
    ;
    (s->val)--;
    spinunlock(&(s->sl));
}

```

→ deadlock,

```

struct semaphore {
    int val;
    spinlock lk;
    list l;
};

sem_init(semaphore *s, int initval) {
    s->val = initval;
    s->sl = 0;
}

block(semaphore *s) {
    listappend(s->l, current);
    spinunlock(&(s->sl));
    schedule();
}

wait(semaphore *s) {
    spinlock(&(s->sl));
    while(s->val <=0) {
        block(s);
    }
    (s->val)--;
    spinunlock(&(s->sl));
}

signal(semaphore *s) {
    spinlock(*(&s->sl));
    (s->val)++;
    x = dequeue(s->sl) and enqueue(readyq, x);
    spinunlock(*(&s->sl));
}

```

→ not holding lock after unblock

Started on Friday, 31 March 2023, 6:18 PM

State Finished

Completed on Friday, 31 March 2023, 7:00 PM

Time taken 41 mins 48 secs

Grade 7.73 out of 15.00 (51.54%)

Question 1

Partially correct

Mark 0.75 out of 1.00

Select all the actions taken by iget()

- a. Panics if inode does not exist in cache
- b. Returns a valid inode if not found in cache ✘
- c. Returns a free-inode , with dev+inode-number set, if not found in cache ✓
- d. Returns the inode with reference count incremented ✓
- e. Returns an inode with given dev+inode-number from cache, if it exists in cache ✓
- f. Returns the inode with inode-cache lock held
- g. Returns the inode locked

Your answer is partially correct.

You have selected too many options.

The correct answers are: Returns an inode with given dev+inode-number from cache, if it exists in cache, Returns the inode with reference count incremented, Returns a free-inode , with dev+inode-number set, if not found in cache

Question 2

Partially correct

Mark 0.60 out of 1.00

Arrange the following in their typical order of use in xv6.

1. use inode
2. iget
3. ilock
4. iunlock
5. iput

Your answer is partially correct.

Grading type: Relative to the next item (including last)

Grade details: 3 / 5 = 60%

Here are the scores for each item in this response:

1. 0 / 1 = 0%
2. 1 / 1 = 100%
3. 0 / 1 = 0%
4. 1 / 1 = 100%
5. 1 / 1 = 100%

The correct order for these items is as follows:

1. iget
2. ilock
3. use inode
4. iunlock
5. iput

Question 3

Incorrect

Mark 0.00 out of 1.00

Note: for this question you get full marks if you select all and only correct options, you get ZERO if at least one option is wrong or not selected.

Select all the correct statements about log structured file systems.

- a. file system recovery recovers all the lost data
- b. xv6 has a log structured file system✓
- c. ext4 is a log structured file system ✗ it's a journaled file system, not log structured
- d. ext2 is by default a log structured file system
- e. log structured file systems considerably improve the recovery time✓

Your answer is incorrect.

The correct answers are: xv6 has a log structured file system, log structured file systems considerably improve the recovery time

Question 4

Correct

Mark 1.00 out of 1.00

Select all the actions taken by ilock()

- a. Get the inode from the inode-cache
- b. Take the sleeplock on the inode, always✓
- c. Lock all the buffers of the file in memory
- d. Take the sleeplock on the inode, optionally
- e. Copy the on-disk inode into in-memory inode, if needed✓
- f. Mark the in-memory inode as valid, if needed✓
- g. Read the inode from disk, if needed✓

Your answer is correct.

The correct answers are: Read the inode from disk, if needed, Copy the on-disk inode into in-memory inode, if needed, Take the sleeplock on the inode, always, Mark the in-memory inode as valid, if needed

Question 5

Incorrect

Mark 0.00 out of 1.00

Maximum size of a file on xv6 in **bytes** is

(just write a numeric answer)

Answer: 16920576



The correct answer is: 71680

Question 6

Partially correct

Mark 1.71 out of 2.00

Select T/F w.r.t physical disk handling in xv6 code

True	False	
<input checked="" type="radio"/>	<input type="radio"/> <input checked="" type="checkbox"/>	The code supports IDE, and not SATA/SCSI
<input type="radio"/> <input checked="" type="checkbox"/>	<input checked="" type="radio"/>	only direct blocks are supported
<input checked="" type="radio"/>	<input type="radio"/> <input checked="" type="checkbox"/>	the superblock does not contain number of free blocks
<input checked="" type="radio"/>	<input type="radio"/> <input checked="" type="checkbox"/>	only 2 disks are handled by default
<input checked="" type="radio"/>	<input checked="" type="radio"/> <input type="checkbox"/>	disk driver handles only one buffer at a time
<input checked="" type="radio"/>	<input type="radio"/> <input checked="" type="checkbox"/>	log is kept on the same device as the file system
<input type="radio"/> <input checked="" type="checkbox"/>	<input checked="" type="radio"/>	device files are not supported

The code supports IDE, and not SATA/SCSI: True

only direct blocks are supported: False

the superblock does not contain number of free blocks: True

only 2 disks are handled by default: True

disk driver handles only one buffer at a time: True

log is kept on the same device as the file system: True

device files are not supported: False

Question 7

Partially correct

Mark 0.50 out of 1.00

Compare XV6 and EXT2 file systems.

Select True/False for each point.

True	False	
<input type="radio"/> <input checked="" type="checkbox"/>	<input checked="" type="radio"/> <input type="checkbox"/>	xv6 contains journal, ext2 does not
<input type="radio"/> <input checked="" type="checkbox"/>	<input type="radio"/> <input checked="" type="checkbox"/>	Ext2 contains superblock but xv6 does not.
<input checked="" type="radio"/> <input type="checkbox"/>	<input type="radio"/> <input checked="" type="checkbox"/>	In both ext2 and xv6, the superblock gives location of first inode block
<input type="radio"/> <input checked="" type="checkbox"/>	<input type="radio"/> <input checked="" type="checkbox"/>	xv6 contains inode bitmap, but ext2 does not
<input checked="" type="radio"/> <input type="checkbox"/>	<input type="radio"/> <input checked="" type="checkbox"/>	Both xv6 and ext2 contain magic number
<input type="radio"/> <input checked="" type="checkbox"/>	<input type="radio"/> <input checked="" type="checkbox"/>	Ext2 contains group descriptors but xv6 does not

xv6 contains journal, ext2 does not: True

Ext2 contains superblock but xv6 does not.: False

In both ext2 and xv6, the superblock gives location of first inode block: False

xv6 contains inode bitmap, but ext2 does not: False

Both xv6 and ext2 contain magic number: False

Ext2 contains group descriptors but xv6 does not: True

Question 8

Correct

Mark 1.00 out of 1.00

An inode is read from disk as a part of this function

- a. iread
- b. readi
- c. iget
- d. sys_read
- e. ilock✓

Your answer is correct.

The correct answer is: ilock

Question 9

Correct

Mark 2.00 out of 2.00

Marks the statements as True/False w.r.t. "struct buf"

True	False	
<input checked="" type="radio"/>	<input type="radio"/> <input checked="" type="checkbox"/>	The reference count (refcnt) in struct buf is = number of processes accessing the buffer
<input checked="" type="radio"/>	<input type="radio"/> <input checked="" type="checkbox"/>	Lock on a buffer is acquired in bget, and released in brelse
<input type="radio"/> <input checked="" type="checkbox"/>	<input checked="" type="radio"/>	The "next" pointer chain gives the buffers in LRU order
<input checked="" type="radio"/>	<input type="radio"/> <input checked="" type="checkbox"/>	B_DIRTY flag means the buffer contains modified data
<input checked="" type="radio"/>	<input type="radio"/> <input checked="" type="checkbox"/>	A buffer can be both on the MRU/LRU list and also on idequeue list.
<input type="radio"/> <input checked="" type="checkbox"/>	<input checked="" type="radio"/>	A buffer can have both B_VALID and B_DIRTY flags set
<input type="radio"/> <input checked="" type="checkbox"/>	<input checked="" type="radio"/>	B_VALID means the buffer is empty and can be reused
<input checked="" type="radio"/>	<input type="radio"/> <input checked="" type="checkbox"/>	The buffers are maintained in LRU order, in the function brelse

The reference count (refcnt) in struct buf is = number of processes accessing the buffer: True

Lock on a buffer is acquired in bget, and released in brelse: True

The "next" pointer chain gives the buffers in LRU order: False

B_DIRTY flag means the buffer contains modified data: True

A buffer can be both on the MRU/LRU list and also on idequeue list.: True

A buffer can have both B_VALID and B_DIRTY flags set: False

B_VALID means the buffer is empty and can be reused: False

The buffers are maintained in LRU order, in the function brelse: True

Question 10

Partially correct

Mark 0.17 out of 1.00

Suppose an application on xv6 does the following:

```
int main() {  
    char arr[128];  
    int fd = open("README", O_RDONLY);  
    read(fd, arr, 100);  
}
```

Assume that the code works.

Which of the following things are true about xv6 kernel code, w.r.t. the above C program.

True False

<input type="radio"/> ✗	<input checked="" type="radio"/> ✗	The loop in readi() will always read a different block using bread()	✗
<input checked="" type="radio"/> ✗	<input type="radio"/> ✗	value of fd will be 3	✓
<input type="radio"/> ✗	<input checked="" type="radio"/> ✗	The "memmove(dst, bp->data + off%BSIZE, m);" in readi() will copy the data from the disk to the kernel buffers	✗
<input checked="" type="radio"/> ✗	<input type="radio"/> ✗	The process will be made to sleep only once	✗
<input checked="" type="radio"/> ✗	<input type="radio"/> ✗	The ONLY function that gets called on return devsw[ip->major].read(ip, dst, n); is consoleread	✗
<input type="radio"/> ✗	<input checked="" type="radio"/> ✗	The data is transferred from disk to kernel buffers first, and then address of arr is mapped to the kernel buffers	✗ No. data is copied into arr.

The loop in readi() will always read a different block using bread(): False
value of fd will be 3: True

The "memmove(dst, bp->data + off%BSIZE, m);" in readi() will copy the data from the disk to the kernel buffers: False

The process will be made to sleep only once: True

The ONLY function that gets called on return devsw[ip->major].read(ip, dst, n); is consoleread: True

The data is transferred from disk to kernel buffers first, and then address of arr is mapped to the kernel buffers: False

Question 11

Not answered

Marked out of 1.00

The lines

```
if(ip->type != T_DIR){  
    iunlockput(ip);  
    return 0;  
}
```

in namex() function

mean

- a. The last path component (which is a file, and not a directory) has been resolved, so release the lock (using iunlockput) and return
- b. No directory entry was found for the file to be opened, hence an error
- c. One of the sub-components on the given path name, was not a directory, hence it's an error
- d. One of the sub-components on the given path name, was a directory, but it was not supposed to be a directory, hence an error
- e. There was a syntax error in the pathname specified
- f. ilock is held on the inode, and hence it's an error if it is a directory
- g. One of the sub-components on the given path name, did not exist, hence it's an error

Your answer is incorrect.

The correct answer is: One of the sub-components on the given path name, was not a directory, hence it's an error

Question 12

Not answered

Marked out of 1.00

Map the function in xv6's file system code, to it's perceived logical layer.

sys_chdir()	Choose...
skipelem	Choose...
ialloc	Choose...
namei	Choose...
bmap	Choose...
filestat()	Choose...
dirlookup	Choose...
balloc	Choose...
ideintr	Choose...
bread	Choose...
stati	Choose...
commit	Choose...

Your answer is incorrect.

The correct answer is: sys_chdir() → system call, skipelem → pathname lookup, ialloc → inode, namei → pathname lookup, bmap → inode, filestat() → file descriptor, dirlookup → directory, balloc → block allocation on disk, ideintr → disk driver, bread → buffer cache, stati → inode, commit → logging

Question 13

Not answered

Marked out of 1.00

Match function with it's functionality

dirlookup	Choose...
namex	Choose...
nameiparent	Choose...
dirlink	Choose...

Your answer is incorrect.

The correct answer is: dirlookup → Search a given name in a given directory, namex → return in-memory inode for a given pathname, nameiparent → return in-memory inode for parent directory of a given pathname, dirlink → Write a new entry in a given directory

[◀ Random Quiz - 5: xv6 make, bootloader, interrupt handling, memory management](#)

Jump to...

(Random Quiz - 7) Pre-Endsem Quiz ►

Started on Thursday, 9 March 2023, 6:20 PM

State Finished

Completed on Thursday, 9 March 2023, 7:23 PM

Time taken 1 hour 3 mins

Overdue 7 mins 42 secs

Grade 5.46 out of 10.00 (54.56%)

Question 1

Partially correct

Mark 0.15 out of 1.00

Consider the following command and its output:

```
$ ls -lht xv6.img kernel
-rw-rw-r-- 1 abhijit abhijit 4.9M Feb 15 11:09 xv6.img
-rwxrwxr-x 1 abhijit abhijit 209K Feb 15 11:09 kernel*
```

Following code in bootmain()

```
readseg((uchar*)elf, 4096, 0);
```

and following selected lines from Makefile

```
xv6.img: bootblock kernel
dd if=/dev/zero of=xv6.img count=10000
dd if=bootblock of=xv6.img conv=notrunc
dd if=kernel of=xv6.img seek=1 conv=notrunc
```

```
kernel: $(OBJS) entry.o entryother initcode kernel.ld
$(LD) $(LDFLAGS) -T kernel.ld -o kernel entry.o $(OBJS) -b binary initcode entryother
$(OBJDUMP) -S kernel > kernel.asm
$(OBJDUMP) -t kernel | sed '1,/SYMBOL TABLE/d; s/ .* //; /^$/d' > kernel.sym
```

Also read the code of bootmain() in xv6 kernel.

Select the options that describe the meaning of these lines and their correlation.

- a. Although the size of the kernel file is 209 Kb, only 4Kb out of it is the actual kernel code and remaining part is all zeroes.
- b. The kernel.asm file is the final kernel file ✗
- c. readseg() reads first 4k bytes of kernel in memory
- d. The kernel is compiled by linking multiple .o files created from .c files; and the entry.o, initcode, entryother files ✓
- e. The bootmain() code does not read the kernel completely in memory
- f. The kernel disk image is ~5MB, the kernel within it is 209 kb, but bootmain() initially reads only first 4kb, and the later part is read using program headers in bootmain().
- g. Although the size of the xv6.img file is ~5MB, only some part out of it is the bootloader+kernel code and remaining part is all zeroes. ✓
- h. The kernel.ld file contains instructions to the linker to link the kernel properly
- i. The kernel disk image is ~5MB, the kernel within it is 209 kb, but bootmain() initially reads only first 4kb, and the later part is not read as it is user programs.

Your answer is partially correct.

You have correctly selected 2.

The correct answers are: The kernel disk image is ~5MB, the kernel within it is 209 kb, but bootmain() initially reads only first 4kb, and the later part is read using program headers in bootmain(), readseg() reads first 4k bytes of kernel in memory, The kernel is compiled by linking multiple .o files created from .c files; and the entry.o, initcode, entryother files, The kernel.ld file contains instructions to the linker to link the kernel properly, Although the size of the xv6.img file is ~5MB, only some part out of it is the bootloader+kernel code and remaining part is all zeroes.

Question 2

Partially correct

Mark 0.20 out of 1.00

For each line of code mentioned on the left side, select the location of sp/esp that is in use

ljmp \$(SEG_KCODE<<3), \$start32
in bootasm.S

0x10000 to 0x7c00



jmp *%eax
in entry.S

Immaterial as the stack is not used here



cli
in bootasm.S

Immaterial as the stack is not used here



readseg((uchar*)elf, 4096, 0);
in bootmain.c

The 4KB area in kernel image, loaded in memory, named as 'stack'



call bootmain
in bootasm.S

0x10000 to 0x7c00



Your answer is partially correct.

You have correctly selected 1.

The correct answer is: ljmp \$(SEG_KCODE<<3), \$start32

in bootasm.S → Immortal as the stack is not used here, jmp *%eax

in entry.S → The 4KB area in kernel image, loaded in memory, named as 'stack', cli

in bootasm.S → Immortal as the stack is not used here, readseg((uchar*)elf, 4096, 0);

in bootmain.c → 0x7c00 to 0, call bootmain

in bootasm.S → 0x7c00 to 0

Question 3

Incorrect

Mark 0.00 out of 1.00

In bootasm.S, on the line

ljmp \$(SEG_KCODE<<3), \$start32

The SEG_KCODE << 3, that is shifting of 1 by 3 bits is done because

- a. The ljmp instruction does a divide by 8 on the first argument
- b. While indexing the GDT using CS, the value in CS is always divided by 8
- c. The value 8 is stored in code segment
- d. The code segment is 16 bit and only lower 13 bits are used for segment number
- e. The code segment is 16 bit and only upper 13 bits are used for segment number

Your answer is incorrect.

The correct answer is: The code segment is 16 bit and only upper 13 bits are used for segment number

Question 4

Correct

Mark 1.00 out of 1.00

What's the trapframe in xv6?

- a. The IDT table
- b. A frame of memory that contains all the trap handler's addresses
- c. A frame of memory that contains all the trap handler code
- d. The sequence of values, including saved registers, constructed on the stack when an interrupt occurs, built by code in trapasm.S only
- e. The sequence of values, including saved registers, constructed on the stack when an interrupt occurs, built by hardware + code in trapasm.S ✓
- f. A frame of memory that contains all the trap handler code's function pointers
- g. The sequence of values, including saved registers, constructed on the stack when an interrupt occurs, built by hardware only

Your answer is correct.

The correct answer is: The sequence of values, including saved registers, constructed on the stack when an interrupt occurs, built by hardware + code in trapasm.S

Question 5

Partially correct

Mark 0.57 out of 1.00

Select all the correct statements about code of bootmain() in xv6

```
void
bootmain(void)
{
    struct elfhdr *elf;
    struct proghdr *ph, *eph;
    void (*entry)(void);
    uchar* pa;

    elf = (struct elfhdr*)0x10000; // scratch space

    // Read 1st page off disk
    readseg((uchar*)elf, 4096, 0);

    // Is this an ELF executable?
    if(elf->magic != ELF_MAGIC)
        return; // let bootasm.S handle error

    // Load each program segment (ignores ph flags).
    ph = (struct proghdr*)((uchar*)elf + elf->phoff);
    eph = ph + elf->phnum;
    for(; ph < eph; ph++){
        pa = (uchar*)ph->paddr;
        readseg(pa, ph->filesz, ph->off);
        if(ph->memsz > ph->filesz)
            stosb(pa + ph->filesz, 0, ph->memsz - ph->filesz);
    }

    // Call the entry point from the ELF header.
    // Does not return!
    entry = (void(*)(void))(elf->entry);
    entry();
}
```

Also, inspect the relevant parts of the xv6 code. binary files, etc and run commands as you deem fit to answer this question.

- a. The readseg finally invokes the disk I/O code using assembly instructions✓
- b. The condition if(ph->memsz > ph->filesz) is never true.
- c. The kernel file in memory is not necessarily a continuously filled in chunk, it may have holes in it.
- d. The kernel ELF file contains actual physical address where particular sections of 'kernel' file should be loaded✓
- e. The elf->entry is set by the linker in the kernel file and it's 8010000
- f. The elf->entry is set by the linker in the kernel file and it's 0x80000000
- g. The kernel file gets loaded at the Physical address 0x10000 +0x80000000 in memory.
- h. The stosb() is used here, to fill in some space in memory with zeroes✓
- i. The kernel file has only two program headers
- j. The kernel file gets loaded at the Physical address 0x10000 in memory.✓
- k. The elf->entry is set by the linker in the kernel file and it's 0x80000000

Your answer is partially correct.

You have correctly selected 4.

The correct answers are: The kernel file gets loaded at the Physical address 0x10000 in memory., The kernel file in memory is not necessarily a continuously filled in chunk, it may have holes in it., The elf->entry is set by the linker in the kernel file and it's 8010000c, The readseg finally invokes the disk I/O code using assembly instructions, The stosb() is used here, to fill in some space in memory with zeroes, The kernel ELF file contains actual physical address where particular sections of 'kernel' file should be loaded, The kernel file has only two program headers

Question 6

Partially correct

Mark 0.50 out of 1.00

Some part of the bootloader of xv6 is written in assembly while some part is written in C. Why is that so?

Select all the appropriate choices

- a. The setting up of the most essential memory management infrastructure needs assembly code
- b. The code in assembly is required for transition to protected mode, from real mode; but calling convention was applicable all the time
- c. The code in assembly is required for transition to protected mode, from real mode; after that calling convention applies, hence code can be written in C ✓
- d. The code for reading ELF file can not be written in assembly

Your answer is partially correct.

You have correctly selected 1.

The correct answers are: The code in assembly is required for transition to protected mode, from real mode; after that calling convention applies, hence code can be written in C, The setting up of the most essential memory management infrastructure needs assembly code

Question 7

Partially correct

Mark 0.50 out of 1.00

For each function/code-point, select the status of segmentation setup in xv6

entry.S

gdt setup with 5 entries (0 to 4) on one processor

✗

kvmalloc() in main()

gdt setup with 5 entries (0 to 4) on one processor

✗

after seginit() in main()

gdt setup with 5 entries (0 to 4) on all processors

✗

after startothers() in main()

gdt setup with 5 entries (0 to 4) on all processors

✓

bootasm.S

gdt setup with 3 entries, at start32 symbol of bootasm.S

✓

bootmain()

gdt setup with 3 entries, at start32 symbol of bootasm.S

✓

Your answer is partially correct.

You have correctly selected 3.

The correct answer is: entry.S → gdt setup with 3 entries, at start32 symbol of bootasm.S, kvmalloc() in main() → gdt setup with 3 entries, at start32 symbol of bootasm.S, after seginit() in main() → gdt setup with 5 entries (0 to 4) on one processor, after startothers() in main() → gdt setup with 5 entries (0 to 4) on all processors, bootasm.S → gdt setup with 3 entries, at start32 symbol of bootasm.S, bootmain() → gdt setup with 3 entries, at start32 symbol of bootasm.S

Question 8

Partially correct

Mark 0.91 out of 1.00

W.r.t. Memory management in xv6,

xv6 uses physical memory upto 224 MB only
Mark statements True or False

True	False	
<input type="radio"/> ✗	<input checked="" type="radio"/> ✗	The switchkvm() call in scheduler() is invoked after control comes to it from swtch() scheduler(), thus demanding execution in new process's context
<input type="radio"/> ✗	<input checked="" type="radio"/> ✗	The switchkvm() call in scheduler() changes CR3 to use page directory of new process
<input checked="" type="radio"/> ✗	<input type="radio"/> ✗	PHYSTOP can be increased to some extent, simply by editing memlayout.h
<input checked="" type="radio"/> ✗	<input type="radio"/> ✗	The switchkvm() call in scheduler() changes CR3 to use page directory kpgdir
<input checked="" type="radio"/> ✗	<input type="radio"/> ✗	The process's address space gets mapped on frames, obtained from ~2MB:224MB range
<input type="radio"/> ✗	<input checked="" type="radio"/> ✗	The kernel's page table given by kpgdir variable is used as stack for scheduler's context
<input checked="" type="radio"/> ✗	<input type="radio"/> ✗	The kernel code and data take up less than 2 MB space
<input checked="" type="radio"/> ✗	<input type="radio"/> ✗	The free page-frame are created out of nearly 222 MB
<input checked="" type="radio"/> ✗	<input type="radio"/> ✗	The stack allocated in entry.S is used as stack for scheduler's context for first processor
<input checked="" type="radio"/> ✗	<input type="radio"/> ✗	The switchkvm() call in scheduler() is invoked after control comes to it from sched(), thus demanding execution in kernel's context
<input checked="" type="radio"/> ✗	<input type="radio"/> ✗	xv6 uses physical memory upto 224 MB only

The switchkvm() call in scheduler() is invoked after control comes to it from swtch() scheduler(), thus demanding execution in new process's context: False

The switchkvm() call in scheduler() changes CR3 to use page directory of new process: False

PHYSTOP can be increased to some extent, simply by editing memlayout.h: True

The switchkvm() call in scheduler() changes CR3 to use page directory kpgdir: True

The process's address space gets mapped on frames, obtained from ~2MB:224MB range: True

The kernel's page table given by kpgdir variable is used as stack for scheduler's context: False

The kernel code and data take up less than 2 MB space: True

The free page-frame are created out of nearly 222 MB: True

The stack allocated in entry.S is used as stack for scheduler's context for first processor: True

The switchkvm() call in scheduler() is invoked after control comes to it from sched(), thus demanding execution in kernel's context: True

xv6 uses physical memory upto 224 MB only: True

Question 9

Partially correct

Mark 0.88 out of 1.00

Select the correct statements about interrupt handling in xv6 code

- a. All the 256 entries in the IDT are filled in xv6 code ✓
- b. The trapframe pointer in struct proc, points to a location on user stack
- c. On any interrupt/syscall/exception the control first jumps in trapasm.S
- d. The CS and EIP are changed immediately (as the first thing) on a hardware interrupt
- e. Each entry in IDT essentially gives the values of CS and EIP to be used in handling that interrupt ✓
- f. The trapframe pointer in struct proc, points to a location on process's kernel stack ✓
- g. xv6 uses the 0x64th entry in IDT for system calls
- h. On any interrupt/syscall/exception the control first jumps in vectors.S ✓
- i. The function trap() is called only in case of hardware interrupt
- j. xv6 uses the 64th entry in IDT for system calls ✓
- k. The function trap() is called even if any of the hardware interrupt/system-call/exception occurs ✓
- l. The CS and EIP are changed only after pushing user code's SS,ESP on stack ✓
- m. Before going to alltraps, the kernel stack contains upto 5 entries.

Your answer is partially correct.

You have correctly selected 7.

The correct answers are: All the 256 entries in the IDT are filled in xv6 code, Each entry in IDT essentially gives the values of CS and EIP to be used in handling that interrupt, xv6 uses the 64th entry in IDT for system calls, On any interrupt/syscall/exception the control first jumps in vectors.S, Before going to alltraps, the kernel stack contains upto 5 entries., The trapframe pointer in struct proc, points to a location on process's kernel stack, The function trap() is called even if any of the hardware interrupt/system-call/exception occurs, The CS and EIP are changed only after pushing user code's SS,ESP on stack

Question 10

Partially correct

Mark 0.75 out of 1.00

```
xv6.img: bootblock kernel
dd if=/dev/zero of=xv6.img count=10000
dd if=bootblock of=xv6.img conv=notrunc
dd if=kernel of=xv6.img seek=1 conv=notrunc
```

Consider above lines from the Makefile. Which of the following is INCORRECT?

- a. Blocks in xv6.img after kernel may be all zeroes.
- b. The size of xv6.img is exactly = (size of bootblock) + (size of kernel) ✓
- c. xv6.img is the virtual processor used by the qemu emulator ✓
- d. The bootblock may be 512 bytes or less (looking at the Makefile instruction)
- e. The kernel is located at block-1 of the xv6.img
- f. The xv6.img is of the size 10,000 blocks of 512 bytes each and occupies 10,000 blocks on the disk.
- g. The size of the xv6.img is nearly 5 MB
- h. The xv6.img is of the size 10,000 blocks of 512 bytes each and occupies upto 10,000 blocks on the disk.
- i. The size of the kernel file is nearly 5 MB ✓
- j. The bootblock is located on block-0 of the xv6.img
- k. The xv6.img is the virtual disk that is created by combining the bootblock and the kernel file.

Your answer is partially correct.

You have correctly selected 3.

The correct answers are: xv6.img is the virtual processor used by the qemu emulator, The xv6.img is of the size 10,000 blocks of 512 bytes each and occupies upto 10,000 blocks on the disk., The size of the kernel file is nearly 5 MB, The size of xv6.img is exactly = (size of bootblock) + (size of kernel)

[◀ Random Quiz 4 : Scheduling, signals, segmentation, paging, compilation, process-state](#)

Jump to...

[Random Quiz - 6 \(xv6 file system\) ▶](#)

Started on Thursday, 16 February 2023, 9:00 PM

State Finished

Completed on Thursday, 16 February 2023, 9:54 PM

Time taken 53 mins 39 secs

Grade 12.88 out of 15.00 (85.86%)

Question 1

Correct

Mark 1.00 out of 1.00

Mark whether the concept is related to scheduling or not.

Yes	No	
<input checked="" type="radio"/>	<input type="radio"/> 	ready-queue
<input checked="" type="radio"/>	<input type="radio"/> 	context-switch
<input checked="" type="radio"/>	<input type="radio"/> 	timer interrupt
<input checked="" type="radio"/>	<input type="radio"/> 	runnable process
<input type="radio"/> 	<input checked="" type="radio"/>	file-table

ready-queue: Yes

context-switch: Yes

timer interrupt: Yes

runnable process: Yes

file-table: No

Question 2

Partially correct

Mark 0.67 out of 1.00

Which of the following parts of a C program do not have any corresponding machine code ?

- a. #directives✓
- b. pointer dereference
- c. typedefs✓
- d. local variable declaration
- e. global variables
- f. function calls
- g. expressions

Your answer is partially correct.

You have correctly selected 2.

The correct answers: #directives, typedefs, global variables

Question 3

Partially correct

Mark 0.50 out of 1.00

Order the sequence of events, in scheduling process P1 after process P0

Control is passed to P1

5	✓
---	---

timer interrupt occurs

4	✗
---	---

context of P1 is loaded from P1's PCB

3	✗
---	---

Process P0 is running

1	✓
---	---

Process P1 is running

6	✓
---	---

context of P0 is saved in P0's PCB

2	✗
---	---

Your answer is partially correct.

You have correctly selected 3.

The correct answer is: Control is passed to P1 → 5, timer interrupt occurs → 2, context of P1 is loaded from P1's PCB → 4, Process P0 is running → 1, Process P1 is running → 6, context of P0 is saved in P0's PCB → 3

Question 4

Partially correct

Mark 0.80 out of 1.00

Mark statements True/False w.r.t. change of states of a process. Note that a statement is true only if the claim and argument both are true.

Reference: The process state diagram (and your understanding of how kernel code works). Note - the diagram does not show zombie state!

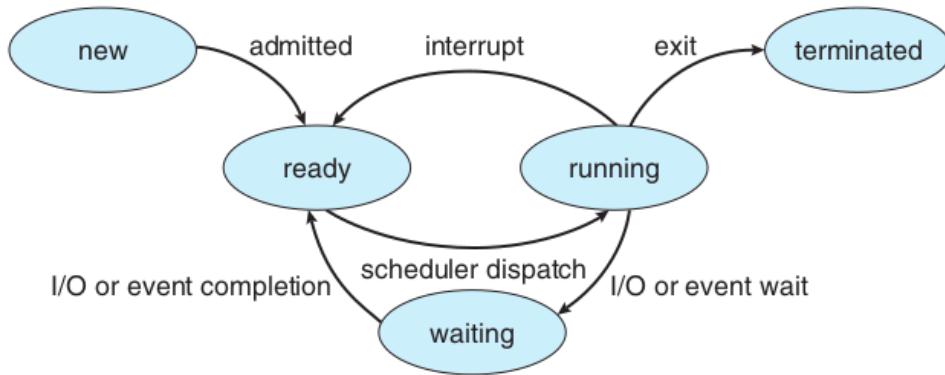


Figure 3.2 Diagram of process state.

True	False		
<input type="radio"/> ✗	<input checked="" type="radio"/> ✗	A process in READY state can not go to WAITING state because the resource on which it will WAIT will not be in use when process is in READY state.	✓
<input checked="" type="radio"/> ✗	<input type="radio"/> ✗	Every forked process has to go through ZOMBIE state, at least for a small duration.	✓
<input checked="" type="radio"/> ✗	<input type="radio"/> ✗	A process in WAITING state can not become RUNNING because the event it's waiting for has not occurred and it has not been moved to ready queue yet	✓
<input type="radio"/> ✗	<input checked="" type="radio"/> ✗	A process only in RUNNING state can become TERMINATED because scheduler moves it to ZOMBIE state first	✗
<input checked="" type="radio"/> ✗	<input type="radio"/> ✗	Only a process in READY state is considered by scheduler	✓

A process in READY state can not go to WAITING state because the resource on which it will WAIT will not be in use when process is in READY state.: False

Every forked process has to go through ZOMBIE state, at least for a small duration.: True

A process in WAITING state can not become RUNNING because the event it's waiting for has not occurred and it has not been moved to ready queue yet: True

A process only in RUNNING state can become TERMINATED because scheduler moves it to ZOMBIE state first: False

Only a process in READY state is considered by scheduler: True

Question 5

Correct

Mark 1.00 out of 1.00

Which of the following are NOT a part of job of a typical compiler?

- a. Invoke the linker to link the function calls with their code, extern globals with their declaration
- b. Process the # directives in a C program
- c. Check the program for syntactical errors
- d. Convert high level language code to machine code
- e. Check the program for logical errors ✓
- f. Suggest alternative pieces of code that can be written ✓

Your answer is correct.

The correct answers are: Check the program for logical errors, Suggest alternative pieces of code that can be written

Question 6

Correct

Mark 1.00 out of 1.00

Select the compiler's view of the process's address space, for each of the following MMU schemes:

(Assume that each scheme,e.g. paging/segmentation/etc is effectively utilised)

Segmentation	many continuous chunks of variable size	✓
Relocation + Limit	one continuous chunk	✓
Paging	one continuous chunk	✓
Segmentation, then paging	many continuous chunks of variable size	✓

Your answer is correct.

The correct answer is: Segmentation → many continuous chunks of variable size, Relocation + Limit → one continuous chunk, Paging → one continuous chunk, Segmentation, then paging → many continuous chunks of variable size

Question 7

Partially correct

Mark 1.56 out of 2.00

Select all the correct statements about the state of a process.

- a. A process waiting for any condition is woken up by another process only
- b. A waiting process starts running after the wait is over
- c. A process that is running is not on the ready queue ✓
- d. A process waiting for I/O completion is typically woken up by the particular interrupt handler code ✓
- e. It is not maintained in the data structures by kernel, it is only for conceptual understanding of programmers
- f. A running process may terminate, or go to wait or become ready again ✓
- g. Typically, it's represented as a number in the PCB
- h. A process can self-terminate only when it's running ✓
- i. Processes in the ready queue are in the ready state ✓
- j. A process changes from running to ready state on a timer interrupt or any I/O wait
- k. A process in ready state is ready to be scheduled ✓
- l. A process in ready state is ready to receive interrupts
- m. Changing from running state to waiting state results in "giving up the CPU"
- n. A process changes from running to ready state on a timer interrupt ✓

Your answer is partially correct.

You have correctly selected 7.

The correct answers are: Typically, it's represented as a number in the PCB, A process in ready state is ready to be scheduled, Processes in the ready queue are in the ready state, A process that is running is not on the ready queue, A running process may terminate, or go to wait or become ready again, A process changes from running to ready state on a timer interrupt, Changing from running state to waiting state results in "giving up the CPU", A process can self-terminate only when it's running, A process waiting for I/O completion is typically woken up by the particular interrupt handler code

Question 8

Correct

Mark 1.00 out of 1.00

Select all the correct statements about zombie processes

Select one or more:

- a. A process becomes zombie when its parent finishes
- b. init() typically keeps calling wait() for zombie processes to get cleaned up ✓
- c. If the parent of a process finishes, before the process itself, then after finishing the process is typically attached to 'init' as parent ✓
- d. A process can become zombie if it finishes, but the parent has finished before it ✓
- e. A process becomes zombie when it finishes, and remains zombie until parent calls wait() on it ✓
- f. Zombie processes are harmless even if OS is up for long time
- g. A zombie process remains zombie forever, as there is no way to clean it up
- h. A zombie process occupies space in OS data structures ✓

Your answer is correct.

The correct answers are: A process becomes zombie when it finishes, and remains zombie until parent calls wait() on it, A process can become zombie if it finishes, but the parent has finished before it, A zombie process occupies space in OS data structures, If the parent of a process finishes, before the process itself, then after finishing the process is typically attached to 'init' as parent, init() typically keeps calling wait() for zombie processes to get cleaned up

Question 9

Partially correct

Mark 0.50 out of 1.00

Select all the correct statements about signals

Select one or more:

- a. The signal handler code runs in kernel mode of CPU
- b. Signals are delivered to a process by another process ✗
- c. Signal handlers once replaced can't be restored
- d. The signal handler code runs in user mode of CPU ✓
- e. Signals are delivered to a process by kernel ✓
- f. SIGKILL definitely kills a process because its code runs in kernel mode of CPU ✗
- g. SIGKILL definitely kills a process because it can't be caught or ignored, and its default action terminates the process ✓
- h. A signal handler can be invoked asynchronously or synchronously depending on signal type ✓

Your answer is partially correct.

You have selected too many options.

The correct answers are: Signals are delivered to a process by kernel, A signal handler can be invoked asynchronously or synchronously depending on signal type, The signal handler code runs in user mode of CPU, SIGKILL definitely kills a process because it can't be caught or ignored, and its default action terminates the process

Question 10

Correct

Mark 1.00 out of 1.00

Map each signal with it's meaning

SIGCHLD	Child Stopped or Terminated	✓
SIGPIPE	Broken Pipe	✓
SIGALRM	Timer Signal from alarm()	✓
SIGUSR1	User Defined Signal	✓
SIGSEGV	Invalid Memory Reference	✓

The correct answer is: SIGCHLD → Child Stopped or Terminated, SIGPIPE → Broken Pipe, SIGALRM → Timer Signal from alarm(), SIGUSR1 → User Defined Signal, SIGSEGV → Invalid Memory Reference

Question 11

Correct

Mark 1.00 out of 1.00

Match the names of PCB structures with kernel

xv6	struct proc	✓
linux	struct task_struct	✓

The correct answer is: xv6 → struct proc, linux → struct task_struct

Question 12

Partially correct

Mark 0.86 out of 1.00

Mark True/False

Statements about scheduling and scheduling algorithms

True False

<input checked="" type="radio"/>	<input type="radio"/> ✗	Generally the voluntary context switches are much more than non-voluntary context switches on a Linux system.	<input checked="" type="checkbox"/>
<input type="radio"/> ✗	<input checked="" type="radio"/>	On Linuxes the CPU utilisation is measured as the time spent in scheduling the idle thread	<input checked="" type="checkbox"/> It's the negation of this. Time NOT spent in idle thread.
<input checked="" type="radio"/>	<input type="radio"/> ✗	A scheduling algorithm is non-preemptive if it does context switch only if a process voluntarily relinquishes CPU or it terminates.	<input checked="" type="checkbox"/>
<input checked="" type="radio"/>	<input type="radio"/> ✗	Statistical observations tell us that most processes have large number of small CPU bursts and relatively smaller numbers of large CPU bursts.	<input checked="" type="checkbox"/>
<input checked="" type="radio"/>	<input type="radio"/> ✗	xv6 code does not care about Processor Affinity	<input checked="" type="checkbox"/>
<input checked="" type="radio"/>	<input type="radio"/> ✗	Response time will be quite poor on non-interruptible kernels	<input checked="" type="checkbox"/>
<input type="radio"/> ✗	<input checked="" type="radio"/>	Processor Affinity refers to memory accesses of a process being stored on cache of that processor	<input checked="" type="checkbox"/>

Generally the voluntary context switches are much more than non-voluntary context switches on a Linux system.: True

On Linuxes the CPU utilisation is measured as the time spent in scheduling the idle thread: False

A scheduling algorithm is non-preemptive if it does context switch only if a process voluntarily relinquishes CPU or it terminates.: True

Statistical observations tell us that most processes have large number of small CPU bursts and relatively smaller numbers of large CPU bursts.: True

xv6 code does not care about Processor Affinity: True

Response time will be quite poor on non-interruptible kernels: True

Processor Affinity refers to memory accesses of a process being stored on cache of that processor: True

Question 13

Correct

Mark 1.00 out of 1.00

Select the state that is not possible after the given state, for a process:

- New: Running ✓
- Ready : Waiting ✓
- Running: : None of these ✓
- Waiting: Running ✓

Question 14

Correct

Mark 1.00 out of 1.00

Which of the following statements is false ?

Select one:

- a. Real time systems generally use non preemptive CPU scheduling. ✓
- b. A process scheduling algorithm is preemptive if the CPU can be forcibly removed from a process.
- c. Time sharing systems generally use preemptive CPU scheduling.
- d. Response time is more predictable in preemptive systems than in non preemptive systems.

Your answer is correct.

The correct answer is: Real time systems generally use non preemptive CPU scheduling.

[◀ Random Quiz - 3 \(processes, memory management, event driven kernel\), compilation-linking-loading, ipc-pipes](#)

Jump to...

[Random Quiz - 5: xv6 make, bootloader, interrupt handling, memory management ►](#)

Started on Thursday, 2 February 2023, 9:00 PM

State Finished

Completed on Thursday, 2 February 2023, 11:00 PM

Time taken 1 hour 59 mins

Grade **14.19** out of 20.00 (**70.93%**)

Question 1

Complete

Mark 0.50 out of 1.00

Select the sequence of events that are NOT possible, assuming an interruptible kernel code

Select one or more:

- a. P1 running
P1 makes system call
Scheduler
P2 running
P2 makes system call and blocks
Scheduler
P1 running again
- b. P1 running
P1 makes system call and blocks
Scheduler
P2 running
P2 makes system call and blocks
Scheduler
P3 running
Hardware interrupt
Interrupt unblocks P1
Interrupt returns
P3 running
Timer interrupt
Scheduler
P1 running
- c. P1 running
P1 makes system call
timer interrupt
Scheduler
P2 running
timer interrupt
Scheduler
P1 running
P1's system call return
- d. P1 running
keyboard hardware interrupt
keyboard interrupt handler running
interrupt handler returns
P1 running
P1 makes system call
system call returns
P1 running
timer interrupt
scheduler
P2 running
- e. P1 running
P1 makes system call
system call returns
P1 running
timer interrupt
Scheduler running
P2 running
- f. P1 running
P1 makes system call and blocks
Scheduler

P2 running
P2 makes system call and blocks
Scheduler
P1 running again

The correct answers are: P1 running
P1 makes system call and blocks
Scheduler
P2 running
P2 makes system call and blocks
Scheduler
P1 running again,
P1 running
P1 makes system call
Scheduler
P2 running
P2 makes system call and blocks
Scheduler
P1 running again

Question 2

Complete

Mark 0.00 out of 1.00

Consider the two programs given below to implement the command (ignore the fact that error checks are not done on return values of functions)

```
$ ls . /tmp/asdfksdf >/tmp/ddd 2>&1
```

Program 1

```
int main(int argc, char *argv[]) {  
    int fd, n, i;  
    char buf[128];  
  
    fd = open("/tmp/ddd", O_WRONLY | O_CREAT, S_IRUSR | S_IWUSR);  
    close(1);  
    dup(fd);  
    close(2);  
    dup(fd);  
    execl("/bin/ls", "/bin/ls", ".", "/tmp/asldjfaldfs", NULL);  
}
```

Program 2

```
int main(int argc, char *argv[]) {  
    int fd, n, i;  
    char buf[128];  
  
    close(1);  
    fd = open("/tmp/ddd", O_WRONLY | O_CREAT, S_IRUSR | S_IWUSR);  
    close(2);  
    fd = open("/tmp/ddd", O_WRONLY | O_CREAT, S_IRUSR | S_IWUSR);  
    execl("/bin/ls", "/bin/ls", ".", "/tmp/asldjfaldfs", NULL);  
}
```

Select all the correct statements about the programs

Select one or more:

- a. Program 1 makes sure that there is one file offset used for '2' and '1'
- b. Program 2 makes sure that there is one file offset used for '2' and '1'
- c. Both program 1 and 2 are incorrect
- d. Program 1 is correct for > /tmp/ddd but not for 2>&1
- e. Program 2 is correct for > /tmp/ddd but not for 2>&1
- f. Program 1 ensures 2>&1 and does not ensure > /tmp/ddd
- g. Program 2 ensures 2>&1 and does not ensure > /tmp/ddd
- h. Program 1 does 1>&2
- i. Only Program 1 is correct
- j. Both programs are correct
- k. Only Program 2 is correct
- l. Program 2 does 1>&2

The correct answers are: Only Program 1 is correct, Program 1 makes sure that there is one file offset used for '2' and '1'

Question 3

Complete

Mark 1.00 out of 1.00

Select the state that is not possible after the given state, for a process:

New: Running

Ready : Waiting

Running: None of these

Waiting: Running

Question 4

Complete

Mark 4.75 out of 5.00

Following code claims to implement the command

/bin/ls -l | /usr/bin/head -3 | /usr/bin/tail -1

Fill in the blanks to make the code work.

Note: Do not include space in writing any option. x[1][2] should be written without any space, and so is the case with [1] or [2]. Pay attention to exact syntax and do not write any extra character like ';' or '=' etc.

```
int main(int argc, char *argv[]) {
```

```
    int pid1, pid2;
```

```
    int pfd[
```

```
    2
```

```
];
```

```
    pipe(
```

```
        pfd[0]
```

```
);
```

```
    pid1 =
```

```
        fork()
```

```
;
```

```
    if(pid1 != 0) {
```

```
        close(pfd[0]
```

```
        [0]
```

```
);
```

```
        close(
```

```
        1
```

```
);
```

```
        dup(
```

```
            pfd[0][1]
```

```
);
```

```
        execl("/bin/ls", "/bin/ls", "
```

```
        -l
```

```
        , NULL);
```

```
    }
```

```
    pipe(
```

```
        pfd[1]
```

```
);
```

```
    pid2
```

```
= fork();
```

```
    if(pid2 == 0) {
```

```
        close(
```

```
            pfd[0][1]
```

```
;
```

```
        close(0);
```

```
        dup(
```

```
            pfd[0][0]
```

```

);
    close(pfd[1]
    [0]
);
    close(
    1
);
    dup(
    pfd[1][1]
);
    execl("/usr/bin/head", "/usr/bin/head", "
-3
", NULL);
} else {
    close(pfd
    [1][1]
);
    close(
    0
);
    dup(
    pfd[1][0]
);
    close(pfd
    [0][0]
);
    execl("/usr/bin/tail", "/usr/bin/tail", "
-1
", NULL);
}
}

```

Question 5

Complete

Mark 1.00 out of 1.00

Select the order in which the various stages of a compiler execute.

Loading	does not exist
Linking	4
Pre-processing	1
Intermediate code generation	3
Syntactical Analysis	2

The correct answer is: Loading → does not exist, Linking → 4, Pre-processing → 1, Intermediate code generation → 3, Syntactical Analysis → 2

Question 6

Complete

Mark 0.00 out of 1.00

Select all the correct statements about named pipes and ordinary(unnamed) pipe

Select one or more:

- a. both named and unnamed pipes require some kind of agreed protocol to be effectively used among multiple processes
- b. named pipes are more efficient than ordinary pipes
- c. named pipe exists even if the processes using it do exit()
- d. ordinary pipe can only be used between related processes
- e. named pipes can be used between multiple processes but ordinary pipes can not be used
- f. a named pipe exists as a file on the file system
- g. named pipe can be used between any processes

The correct answers are: ordinary pipe can only be used between related processes, named pipe can be used between any processes, a named pipe exists as a file on the file system, named pipe exists even if the processes using it do exit(), both named and unnamed pipes require some kind of agreed protocol to be effectively used among multiple processes

Question 7

Complete

Mark 0.33 out of 1.00

Select the sequence of events that are NOT possible, assuming a non-interruptible kernel code

(Note: non-interruptible kernel code means, if the kernel code is executing, then interrupts will be disabled).

Note: A possible sequence may have some missing steps in between. An impossible sequence will have n and n+1th steps such that n+1th step can not follow n'th step.

Select one or more:

- a. P1 running
P1 makes system call
system call returns
P1 running
timer interrupt
Scheduler running
P2 running
- b. P1 running
P1 makes system call and blocks
Scheduler
P2 running
P2 makes system call and blocks
Scheduler
P3 running
Hardware interrupt
Interrupt unblocks P1
Interrupt returns
P3 running
Timer interrupt
Scheduler
P1 running
- c. P1 running
P1 makes system call
timer interrupt
Scheduler
P2 running
timer interrupt
Scheduler
P1 running
P1's system call return
- d. P1 running
P1 makes system call and blocks
Scheduler
P2 running
P2 makes system call and blocks
Scheduler
P1 running again
- e. P1 running
keyboard hardware interrupt
keyboard interrupt handler running
interrupt handler returns
P1 running
P1 makes system call
system call returns
P1 running
timer interrupt
scheduler
P2 running

f.

P1 running
P1 makes system call
Scheduler
P2 running
P2 makes system call and blocks
Scheduler
P1 running again

The correct answers are: P1 running

P1 makes system call and blocks

Scheduler

P2 running

P2 makes system call and blocks

Scheduler

P1 running again, P1 running

P1 makes system call

timer interrupt

Scheduler

P2 running

timer interrupt

Scheuler

P1 running

P1's system call return,

P1 running

P1 makes system call

Scheduler

P2 running

P2 makes system call and blocks

Scheduler

P1 running again

Question 8

Complete

Mark 1.00 out of 1.00

Which of the following are NOT a part of job of a typical compiler?

- a. Convert high level language code to machine code
- b. Process the # directives in a C program
- c. Check the program for logical errors
- d. Check the program for syntactical errors
- e. Suggest alternative pieces of code that can be written
- f. Invoke the linker to link the function calls with their code, extern globals with their declaration

The correct answers are: Check the program for logical errors, Suggest alternative pieces of code that can be written

Question 9

Complete

Mark 1.40 out of 2.00

Match the elements of C program to their place in memory

Allocated Memory	Heap
Local Static variables	Stack
#include files	Code
Global variables	Data
#define MACROS	No memory needed
Global Static variables	Data
Function code	Code
Code of main()	Code
Local Variables	Stack
Arguments	Stack

The correct answer is: Allocated Memory → Heap, Local Static variables → Stack, #include files → No memory needed, Global variables → Data, #define MACROS → No Memory needed, Global Static variables → Data, Function code → Code, Code of main() → Code, Local Variables → Stack, Arguments → Stack

Question 10

Complete

Mark 0.67 out of 1.00

Select all the correct statements about zombie processes

Select one or more:

- a. If the parent of a process finishes, before the process itself, then after finishing the process is typically attached to 'init' as parent
- b. A process becomes zombie when it finishes, and remains zombie until parent calls wait() on it
- c. A process becomes zombie when its parent finishes
- d. A zombie process occupies space in OS data structures
- e. A process can become zombie if it finishes, but the parent has finished before it
- f. A zombie process remains zombie forever, as there is no way to clean it up
- g. Zombie processes are harmless even if OS is up for long time
- h. init() typically keeps calling wait() for zombie processes to get cleaned up

The correct answers are: A process becomes zombie when it finishes, and remains zombie until parent calls wait() on it, A process can become zombie if it finishes, but the parent has finished before it, A zombie process occupies space in OS data structures, If the parent of a process finishes, before the process itself, then after finishing the process is typically attached to 'init' as parent, init() typically keeps calling wait() for zombie processes to get cleaned up

Question 11

Complete

Mark 0.29 out of 1.00

Order the events that occur on a timer interrupt:

Jump to a code pointed by IDT

2

Set the context of the new process

5

Change to kernel stack of currently running process

6

Jump to scheduler code

3

Save the context of the currently running process

1

Execute the code of the new process

7

Select another process for execution

4

The correct answer is: Jump to a code pointed by IDT → 2, Set the context of the new process → 6, Change to kernel stack of currently running process → 1, Jump to scheduler code → 4, Save the context of the currently running process → 3, Execute the code of the new process → 7, Select another process for execution → 5

Question 12

Complete

Mark 1.00 out of 1.00

Consider the following code and MAP the file to which each fd points at the end of the code.

```
int main(int argc, char *argv[]) {  
    int fd1, fd2 = 1, fd3 = 1, fd4 = 1;  
  
    fd1 = open("/tmp/1", O_WRONLY | O_CREAT, S_IRUSR|S_IWUSR);  
    fd2 = open("/tmp/2", O_RDONLY);  
    fd3 = open("/tmp/3", O_WRONLY | O_CREAT, S_IRUSR|S_IWUSR);  
    close(0);  
    close(1);  
    dup(fd2);  
    dup(fd3);  
    close(fd3);  
    dup2(fd2, fd4);  
    printf("%d %d %d %d\n", fd1, fd2, fd3, fd4);  
    return 0;  
}
```

fd1	/tmp/1
fd2	/tmp/2
fd4	/tmp/2
0	/tmp/2
2	stderr
1	/tmp/3
fd3	closed

The correct answer is: fd1 → /tmp/1, fd2 → /tmp/2, fd4 → /tmp/2, 0 → /tmp/2, 2 → stderr, 1 → /tmp/3, fd3 → closed

Question 13

Complete

Mark 0.50 out of 1.00

Select the compiler's view of the process's address space, for each of the following MMU schemes:
(Assume that each scheme,e.g. paging/segmentation/etc is effectively utilised)

Paging	Many continuous chunks of same size
Segmentation, then paging	Many continuous chunks each of page size
Relocation + Limit	one continuous chunk
Segmentation	many continuous chunks of variable size

The correct answer is: Paging → one continuous chunk, Segmentation, then paging → many continuous chunks of variable size, Relocation + Limit → one continuous chunk, Segmentation → many continuous chunks of variable size

Question 14

Complete

Mark 0.75 out of 1.00

Consider the image given below, which explains how paging works.

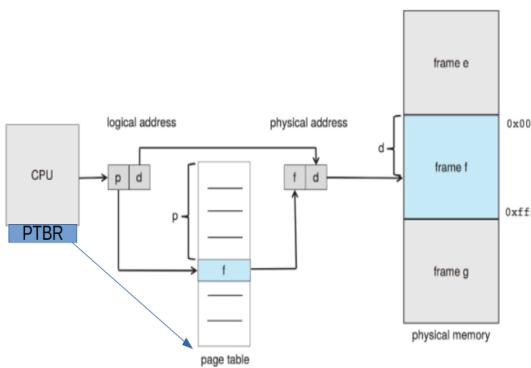


Figure 9.8 Paging hardware.

Mention whether each statement is True or False, with respect to this image.

True False

- Size of page table is always determined by the size of RAM
- The PTBR is present in the CPU as a register
- The page table is indexed using page number
- The page table is indexed using frame number
- Maximum Size of page table is determined by number of bits used for page number
- The physical address may not be of the same size (in bits) as the logical address
- The page table is itself present in Physical memory
- The locating of the page table using PTBR also involves paging translation

Size of page table is always determined by the size of RAM: False

The PTBR is present in the CPU as a register: True

The page table is indexed using page number: True

The page table is indexed using frame number: False

Maximum Size of page table is determined by number of bits used for page number: True

The physical address may not be of the same size (in bits) as the logical address: True

The page table is itself present in Physical memory: True

The locating of the page table using PTBR also involves paging translation: False

Question 15

Complete

Mark 1.00 out of 1.00

A process blocks itself means

- a. The kernel code of system call calls scheduler
- b. The kernel code of an interrupt handler, moves the process to a waiting queue and calls scheduler
- c. The application code calls the scheduler
- d. The kernel code of system call, called by the process, moves the process to a waiting queue and calls scheduler

The correct answer is: The kernel code of system call, called by the process, moves the process to a waiting queue and calls scheduler

[◀ Random Quiz - 2: bootloader, system calls, fork-exec, open-read-write, linux-basics, processes](#)

Jump to...

[Random Quiz 4 : Scheduling, signals, segmentation, paging, compilation, process-state ►](#)

Started on Monday, 16 January 2023, 9:00 PM

State Finished

Completed on Monday, 16 January 2023, 10:05 PM

Time taken 1 hour 4 mins

Grade 11.52 out of 15.00 (76.78%)

Question 1

Correct

Mark 1.00 out of 1.00

Is the terminal a part of the kernel on GNU/Linux systems?

- a. yes
- b. no ✓ wrong

The correct answer is: no

Question 2

Partially correct

Mark 0.67 out of 1.00

Select all the correct statements about bootloader.

Every wrong selection will deduct marks proportional to $1/n$ where n is total wrong choices in the question.

You will get minimum a zero.

- a. Bootloaders allow selection of OS to boot from ✓
- b. LILO is a bootloader ✓
- c. The bootloader loads the BIOS
- d. Modern Bootloaders often allow configuring the way an OS boots
- e. Bootloader must be one sector in length

Your answer is partially correct.

You have correctly selected 2.

The correct answers are: LILO is a bootloader, Modern Bootloaders often allow configuring the way an OS boots, Bootloaders allow selection of OS to boot from

Question 3

Correct

Mark 2.00 out of 2.00

What will this program do?

```
int main() {  
    fork();  
    execl("/bin/ls", "/bin/ls", NULL);  
    printf("hello");  
}
```

- a. run ls twice✓
- b. run ls twice and print hello twice
- c. run ls once
- d. one process will run ls, another will print hello
- e. run ls twice and print hello twice, but output will appear in some random order

Your answer is correct.

The correct answer is: run ls twice

Question 4

Correct

Mark 1.00 out of 1.00

Compare multiprogramming with multitasking

- a. A multitasking system is not necessarily multiprogramming
- b. A multiprogramming system is not necessarily multitasking✓

The correct answer is: A multiprogramming system is not necessarily multitasking

Question 5

Correct

Mark 1.00 out of 1.00

A process blocks itself means

- a. The kernel code of system call, called by the process, moves the process to a waiting queue and calls scheduler✓
- b. The application code calls the scheduler
- c. The kernel code of an interrupt handler, moves the process to a waiting queue and calls scheduler
- d. The kernel code of system call calls scheduler

The correct answer is: The kernel code of system call, called by the process, moves the process to a waiting queue and calls scheduler

Question 6

Correct

Mark 1.00 out of 1.00

which of the following is not a difference between real mode and protected mode

- a. in real mode the segment is multiplied by 16, in protected mode segment is used as index in GDT
- b. processor starts in real mode
- c. in real mode the addressable memory is more than in protected mode✓
- d. in real mode general purpose registers are 16 bit, in protected mode they are 32 bit
- e. in real mode the addressable memory is less than in protected mode

The correct answer is: in real mode the addressable memory is more than in protected mode

Question 7

Correct

Mark 1.00 out of 1.00

When you turn your computer ON, you are often shown an option like "Press F9 for boot options". What does this mean?

- a. The choice of booting slowly or fast
- b. The choice of the boot loader (e.g. GRUB or Windows-Loader)
- c. The BIOS allows us to choose the boot device, the device from which the boot loader will be loaded✓
- d. The choice of which OS to boot from

The correct answer is: The BIOS allows us to choose the boot device, the device from which the boot loader will be loaded

Question 8

Partially correct

Mark 0.83 out of 1.00

Select the correct statements about hard and soft links

Select one or more:

- a. Deleting a hard link always deletes the file
- b. Deleting a soft link deletes only the actual file
- c. Soft links increase the link count of the actual file inode
- d. Hard links increase the link count of the actual file inode✓
- e. Deleting a soft link deletes both the link and the actual file
- f. Deleting a hard link deletes the file, only if link count was 1✓
- g. Hard links enforce separation of filename from its metadata in on-disk data structures.
- h. Soft links can span across partitions while hard links can't✓
- i. Hard links can span across partitions while soft links can't
- j. Hard links share the inode✓
- k. Soft link shares the inode of actual file
- l. Deleting a soft link deletes the link, not the actual file✓

Your answer is partially correct.

You have correctly selected 5.

The correct answers are: Soft links can span across partitions while hard links can't, Hard links increase the link count of the actual file inode, Deleting a soft link deletes the link, not the actual file, Deleting a hard link deletes the file, only if link count was 1, Hard links share the inode, Hard links enforce separation of filename from its metadata in on-disk data structures.

Question 9

Correct

Mark 1.00 out of 1.00

Consider the following programs

[exec1.c](#)

```
#include <unistd.h>
#include <stdio.h>
int main() {
    execl("./exec2", "./exec2", NULL);
}
```

[exec2.c](#)

```
#include <unistd.h>
#include <stdio.h>
int main() {
    execl("/bin/ls", "/bin/ls", NULL);
    printf("hello\n");
}
```

Compiled as

```
cc  exec1.c -o exec1
cc  exec2.c -o exec2
```

And run as

```
$ ./exec1
```

Explain the output of the above command (./exec1)

Assume that /bin/ls , i.e. the 'ls' program exists.

Select one:

- a. Execution fails as one exec can't invoke another exec
- b. "ls" runs on current directory✓
- c. Execution fails as the call to execl() in exec2 fails
- d. Program prints hello
- e. Execution fails as the call to execl() in exec1 fails

Your answer is correct.

The correct answer is: "ls" runs on current directory

Question 10

Partially correct

Mark 0.60 out of 1.00

Select all the correct statements about two modes of CPU operation

Select one or more:

- a. There is an instruction like 'iret' to return from kernel mode to user mode ✓
- b. The software interrupt instructions change the mode from user mode to kernel mode and jumps to predefined location simultaneously ✓
- c. The two modes are essential for a multiprogramming system
- d. The two modes are essential for a multitasking system
- e. Some instructions are allowed to run only in user mode, while all instructions can run in kernel mode ✓

Your answer is partially correct.

You have correctly selected 3.

The correct answers are: The two modes are essential for a multiprogramming system, The two modes are essential for a multitasking system, There is an instruction like 'iret' to return from kernel mode to user mode, The software interrupt instructions change the mode from user mode to kernel mode and jumps to predefined location simultaneously, Some instructions are allowed to run only in user mode, while all instructions can run in kernel mode

Question 11

Partially correct

Mark 0.67 out of 1.00

Order the following events in boot process (from 1 onwards)

Boot loader	2	✓
BIOS	1	✓
Login interface	6	✗
Init	4	✓
Shell	5	✗
OS	3	✓

Your answer is partially correct.

You have correctly selected 4.

The correct answer is: Boot loader → 2, BIOS → 1, Login interface → 5, Init → 4, Shell → 6, OS → 3

Question 12

Partially correct

Mark 0.75 out of 3.00

Select correct statements about mounting

Select one or more:

- a. Even in operating systems with a pluggable kernel module for file systems, the code for mounting any particular file system must be already present in the operating system system kernel
- b. The mount point must be a directory
- c. Mounting deletes all data at the mount-point
- d. Mounting makes all disk partitions available as one name space
- e. On Linuxes mounting can be done only while booting the OS
- f. It's possible to mount a partition on one computer, into namespace of another computer. ✓
- g. Mounting is attaching a disk-partition with a filesystem on it, into another file system name-space ✓
- h. The existing name-space at the mount-point is no longer visible after mounting
- i. The mount point can be a file as well ✗
- j. In operating systems with a pluggable kernel module for file systems, the code for mounting a particular file system is provided by the module of that file system. ✓

Your answer is partially correct.

You have correctly selected 3.

The correct answers are: Mounting is attaching a disk-partition with a filesystem on it, into another file system name-space, The mount point must be a directory, The existing name-space at the mount-point is no longer visible after mounting, Mounting makes all disk partitions available as one name space, In operating systems with a pluggable kernel module for file systems, the code for mounting a particular file system is provided by the module of that file system., It's possible to mount a partition on one computer, into namespace of another computer.

[◀ Random Quiz - 1 \(Pre-Requisite Quiz\)](#)

Jump to...

[Random Quiz - 3 \(processes, memory management, event driven kernel\), compilation-linking-loading, ipc-pipes ►](#)

Started on Friday, 17 March 2023, 2:29 PM

State Finished

Completed on Friday, 17 March 2023, 4:33 PM

Time taken 2 hours 3 mins

Grade 6.75 out of 10.00 (67.46%)

Question 1

Correct

Mark 0.50 out of 0.50

The struct buf has a sleeplock, and not a spinlock, because

- a. sleeplock is preferable because it is used in interrupt context and spinlock can not be used in interrupt context
- b. struct buf is used as a general purpose cache by kernel and cache operations take lot of time, so better to use sleeplock rather than spinlock
- c. It could be a spinlock, but xv6 has chosen sleeplock for purpose of demonstrating how to use a sleeplock.
- d. struct buf is used for disk I/o which takes lot of time, so sleeping/blocking is preferred to spinning/busy-wait for the desired buf. ✓
- e. struct buf is used for disk I/o which takes lot of time, so sleeping/blocking is the only option available.

Your answer is correct.

The correct answer is: struct buf is used for disk I/o which takes lot of time, so sleeping/blocking is preferred to spinning/busy-wait for the desired buf.

Question 2

Partially correct

Mark 0.91 out of 1.00

Given below is code of sleeplock in xv6.

```
// Long-term locks for processes
struct sleeplock {
    uint locked;          // Is the lock held?
    struct spinlock lk;  // spinlock protecting this sleep lock

    // For debugging:
    char *name;           // Name of lock.
    int pid;              // Process holding lock
};
```

```
void
acquiresleep(struct sleeplock *lk)
{
    acquire(&lk->lk);
    while (lk->locked) {
        sleep(lk, &lk->lk);
    }
    lk->locked = 1;
    lk->pid = myproc()->pid;
    release(&lk->lk);
}
```

```
void
releasesleep(struct sleeplock *lk)
{
    acquire(&lk->lk);
    lk->locked = 0;
    lk->pid = 0;
    wakeup(lk);
    release(&lk->lk);
}
```

Mark the statements as True/False w.r.t. this code.

True	False	
<input checked="" type="radio"/>	<input type="radio"/>	The spinlock lk->lk is held when the process comes out of sleep()
<input type="radio"/>	<input checked="" type="radio"/>	sleep() is called holding a spinlock. This could be avoided by releasing the lock before calling sleep() and acquiring it again after call to sleep()
<input type="radio"/>	<input checked="" type="radio"/>	acquire(&lk->lk); while (!lk->locked) { sleep(lk, &lk->lk); } could also be written as acquire(&lk->lk); if (!lk->locked) { sleep(lk, &lk->lk); }
<input checked="" type="radio"/>	<input type="radio"/>	All processes waiting for the sleeplock will have a race for aquiring lk->lk spinlock, because all are woken up

True False

<input type="radio"/> <input checked="" type="checkbox"/>	<input checked="" type="checkbox"/> <input type="radio"/>	the 'spinlock lk' protects 'locked' variable, but not the 'name' nor the 'pid'	<input checked="" type="checkbox"/>
<input checked="" type="checkbox"/> <input type="radio"/>	<input type="radio"/>	sleep() is the function which blocks a process.	<input checked="" type="checkbox"/>
<input checked="" type="checkbox"/> <input type="radio"/>	<input type="radio"/>	the 'spinlock lk' is needed in a sleeplock, because access to the sleeplock for locking/unlocking itself creates a critical section	<input checked="" type="checkbox"/>
<input checked="" type="checkbox"/> <input type="radio"/>	<input type="radio"/>	Sleeplock() will ensure that either the process gets the lock or the process gets blocked.	<input checked="" type="checkbox"/>
<input type="radio"/> <input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	Wakeup() will wakeup the first process waiting for the lock	<input checked="" type="checkbox"/> Wakeup() will wakeup all processes waiting for the lock
<input type="radio"/> <input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	A process has acquired the sleeplock when it comes out of sleep()	<input checked="" type="checkbox"/>
<input checked="" type="checkbox"/> <input type="radio"/>	<input type="radio"/>	The process which called acquiresleep() and then got blocked, is woken up by the timer interrupt	<input checked="" type="checkbox"/> it's woken up by another process which called releasesleep() and then wakeup()

The spinlock lk->lk is held when the process comes out of sleep(): True

sleep() is called holding a spinlock. This could be avoided by releasing the lock before calling sleep() and acquiring it again after call to sleep():

False

```
acquire(&lk->lk);
while (!lk->locked) {
    sleep(lk, &lk->lk);
}
```

could also be written as

```
acquire(&lk->lk);
if (!lk->locked) {
    sleep(lk, &lk->lk);
}
```

: False

All processes waiting for the sleeplock will have a race for aquiring lk->lk spinlock, because all are woken up: True

the 'spinlock lk' protects 'locked' variable, but not the 'name' nor the 'pid': False

sleep() is the function which blocks a process.: True

the 'spinlock lk' is needed in a sleeplock, because access to the sleeplock for locking/unlocking itself creates a critical section: True

Sleeplock() will ensure that either the process gets the lock or the process gets blocked.: True

Wakeup() will wakeup the first process waiting for the lock: False

A process has acquired the sleeplock when it comes out of sleep(): False

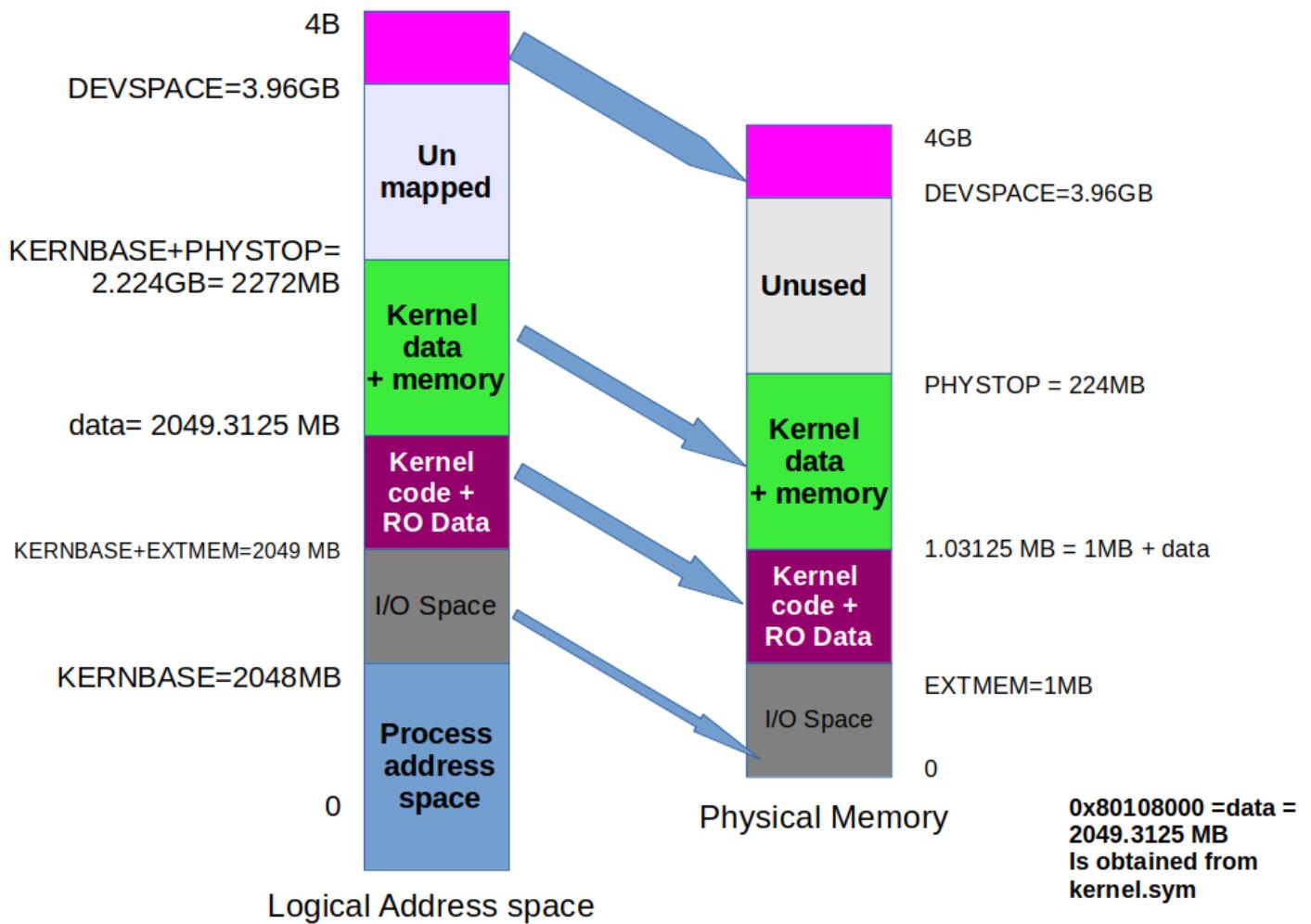
The process which called acquiresleep() and then got blocked, is woken up by the timer interrupt: True

Question 3

Partially correct

Mark 0.36 out of 0.50

With respect to this diagram, mark statements as True/False.



True False

<input checked="" type="radio"/>	<input type="radio"/>	This diagram only shows the absolutely defined virtual->physical mappings, not the mappings defined at run time by kernel.	<input checked="" type="checkbox"/>
<input checked="" type="radio"/>	<input type="radio"/>	The kernel virtual addresses start from KERNLINK = KERNBASE + EXTMEM	<input checked="" type="checkbox"/>
<input checked="" type="radio"/>	<input type="radio"/>	PHYSTOP can be changed , but that needs kernel recompilation and re-execution.	<input checked="" type="checkbox"/>
<input type="radio"/>	<input checked="" type="radio"/>	"Kernel data + memory" on right side, here refers to the region from which pages are allocated to the kernel and process both.	<input checked="" type="checkbox"/> "Kernel data + memory" on LEFT side, here refers to the virtual addresses of kernel used at run time.
<input checked="" type="radio"/>	<input checked="" type="radio"/>	The process's pages are mapped into physical memory from 1.03125 MB to PHYSTOP.	<input checked="" type="checkbox"/>
<input checked="" type="radio"/>	<input checked="" type="radio"/>	When bootloader loads the kernel, then physical memory from EXTMEM upto EXTMEM + data is occupied.	<input checked="" type="checkbox"/>

True **False**



The kernel file, after compilation, has maximum virtual address up to "data" as shown in the diagram, which is equal to "end" variable



This diagram only shows the absolutely defined virtual->physical mappings, not the mappings defined at run time by kernel.: True

The kernel virtual addresses start from KERNLINK = KERNBASE + EXTMEM: True

PHYSTOP can be changed , but that needs kernel recompilation and re-execution.: True

"Kernel data + memory" on right side, here refers to the region from which pages are allocated to the kernel and process both.: True

The process's pages are mapped into physical memory from 1.03125 MB to PHYSTOP.: True

When bootloader loads the kernel, then physical memory from EXTMEM upto EXTMEM + data is occupied.: True

The kernel file, after compilation, has maximum virtual address up to "data" as shown in the diagram, which is equal to "end" variable: True

Question 4

Incorrect

Mark 0.00 out of 0.25

Select the odd one out

- a. Kernel stack of new process to Process stack of new process
- b. Process stack of running process to kernel stack of running process
- c. Kernel stack of scheduler to kernel stack of new process
- d. Kernel stack of running process to kernel stack of scheduler
- e. Kernel stack of new process to kernel stack of scheduler

The correct answer is: Kernel stack of new process to kernel stack of scheduler

Question 5

Partially correct

Mark 0.80 out of 1.00

Mark the statements as True/False w.r.t. swtch()

True	False	
<input checked="" type="radio"/>	<input type="radio"/>	swtch() is written in assembly language, because it violates calling convention, by changing the stack itself.
<input checked="" type="radio"/>	<input type="radio"/>	push in swtch() happens on old stack, while pop happens from new stack
<input type="radio"/>	<input checked="" type="radio"/>	swtch() called from scheduler() changes the stack from the process's kernel stack to the scheduler's kernel stack.
<input type="radio"/>	<input checked="" type="radio"/>	switch stores the old context on new stack, and restores new context from old stack.
<input checked="" type="radio"/>	<input type="radio"/>	p->context used in scheduler()->swtch() was Generally set when the process was interrupted earlier, and came via sched()->swtch()
<input checked="" type="radio"/>	<input type="radio"/>	swtch() changes the context from "old" to "new"
<input type="radio"/>	<input checked="" type="radio"/>	sched() is the only place when p->context is set
<input type="radio"/>	<input checked="" type="radio"/>	swtch() is written in assembly language because it violates the calling convention by pushing parameters on the stack on its own.
<input type="radio"/>	<input checked="" type="radio"/>	movl %esp, (%eax) means, *(c->scheduler) = contents of esp When swtch() is called from scheduler()
<input checked="" type="radio"/>	<input type="radio"/>	swtch() is called only from sched() or scheduler()

swtch() is written in assembly language, because it violates calling convention, by changing the stack itself.: True

push in swtch() happens on old stack, while pop happens from new stack: True

swtch() called from scheduler() changes the stack from the process's kernel stack to the scheduler's kernel stack.: False

switch stores the old context on new stack, and restores new context from old stack.: False

p->context used in scheduler()->swtch() was **Generally** set when the process was interrupted earlier, and came via sched()->swtch(): True

swtch() changes the context from "old" to "new": True

sched() is the only place when p->context is set: False

swtch() is written in assembly language because it violates the calling convention by pushing parameters on the stack on its own.: False

movl %esp, (%eax)

means, *(c->scheduler) = contents of esp

When swtch() is called from scheduler(): False

swtch() is called only from sched() or scheduler(): True

Question 6

Partially correct

Mark 0.44 out of 0.50

Mark the statements as True/False, with respect to the use of the variable "chan" in struct proc.

True	False	
<input type="radio"/> <input checked="" type="checkbox"/>	when chan is NULL, the 'state' in proc must be RUNNABLE.	✓
<input checked="" type="checkbox"/> <input type="radio"/>	in xv6, the address of an appropriate variable is used as a "condition" for a waiting process.	✓
<input checked="" type="checkbox"/> <input type="checkbox"/>	When chan is not NULL, the 'state' in struct proc must be SLEEPING	✗
<input type="radio"/> <input checked="" type="checkbox"/>	Changing the state of a process automatically changes the value of 'chan'	✓
<input type="radio"/> <input checked="" type="checkbox"/>	chan is the head pointer to a linked list of processes, waiting for a particular event to occur	✓
<input checked="" type="checkbox"/> <input type="radio"/>	chan stores the address of the variable, representing a condition, for which the process is waiting.	✓
<input checked="" type="checkbox"/> <input type="radio"/>	The value of 'chan' is changed only in sleep()	✓
<input checked="" type="checkbox"/> <input type="radio"/>	'chan' is used only by the sleep() and wakeup1() functions.	✓

when chan is NULL, the 'state' in proc must be RUNNABLE.: False

in xv6, the address of an appropriate variable is used as a "condition" for a waiting process.: True

When chan is not NULL, the 'state' in struct proc must be SLEEPING: True

Changing the state of a process automatically changes the value of 'chan': False

chan is the head pointer to a linked list of processes, waiting for a particular event to occur: False

chan stores the address of the variable, representing a condition, for which the process is waiting.: True

The value of 'chan' is changed only in sleep(): True

'chan' is used only by the sleep() and wakeup1() functions.: True

Question 7

Partially correct

Mark 0.15 out of 0.25

Match function with its meaning

ideintr	disk interrupt handler, transfer data from controller to buffer, wake up processes waiting for this buffer, start I/O for next buffer	✓
idewait	Wait for disc controller to be ready	✓
ideinit	Initialize the disc controller	✓
iderw	tell disc controller to complete I/O for all pending requests	✗
idestart	Issue a disk read/write for a buffer, block the issuing process	✗

Your answer is partially correct.

You have correctly selected 3.

The correct answer is: ideintr → disk interrupt handler, transfer data from controller to buffer, wake up processes waiting for this buffer, start I/O for next buffer, idewait → Wait for disc controller to be ready, ideinit → Initialize the disc controller, iderw → Issue a disk read/write for a buffer, block the issuing process, idestart → tell disc controller to start I/O for the first buffer on idequeue

Question 8

Partially correct

Mark 0.25 out of 0.50

when is each of the following stacks allocated?

kernel stack of process	during fork() in allocproc()	✓
kernel stack for scheduler, on first processor	in entry.S	✓
user stack of process	during exec()	✗
kernel stack for the scheduler, on other processors	in entry.S	✗

Your answer is partially correct.

You have correctly selected 2.

The correct answer is: kernel stack of process → during fork() in allocproc(), kernel stack for scheduler, on first processor → in entry.S, user stack of process → during fork() in copyuvm(), kernel stack for the scheduler, on other processors → in main()->startothers()

Question 9

Correct

Mark 0.25 out of 0.25

Which of the following is not a task of the code of swtch() function

- a. Load the new context
- b. Save the return value of the old context code ✓
- c. Jump to next context EIP ✗
- d. Change the kernel stack location ✓
- e. Switch stacks
- f. Save the old context

The correct answers are: Save the return value of the old context code, Change the kernel stack location

Question 10

Correct

Mark 0.25 out of 0.25

The variable 'end' used as argument to kinit1 has the value

- a. 80102da0
- b. 801154a8✓
- c. 81000000
- d. 8010a48c
- e. 80110000
- f. 80000000

The correct answer is: 801154a8

Question 11

Partially correct

Mark 0.23 out of 0.50

Which of the following is DONE by allocproc() ?

- a. setup the trapframe and context pointers appropriately✓
- b. setup the contents of the trapframe of the process properly✗
- c. allocate PID to the process
- d. ensure that the process starts in forkret()✓
- e. allocate kernel stack for the process✓
- f. ensure that the process starts in trapret()
- g. Select an UNUSED struct proc for use✓
- h. setup kernel memory mappings for the process

The correct answers are: Select an UNUSED struct proc for use, allocate PID to the process, allocate kernel stack for the process, setup the trapframe and context pointers appropriately, ensure that the process starts in forkret()

Question 12

Incorrect

Mark 0.00 out of 0.50

The first instruction that runs when you do "make qemu" is

cli

from bootasm.S

Why?

- a. "cli" clears all registers and makes them zero, so that processor is as good as "new"
- b. It disables interrupts. It is required because the interrupt handlers of kernel are not yet installed.
- c. "cli" that is Command Line Interface needs to be enabled first
- d. "cli" enables interrupts, it is required because the kernel supports interrupts.
- e. "cli" stands for clear screen and the screen should be cleared before OS boots.
- f. "cli" clears the pipeline of the CPU so that it is as good as "fresh" CPU
- g. "cli" disables interrupts. It is required because as of now there are no interrupt handlers available X
- h. "cli" enables interrupts, it is required because the kernel must handle interrupts.

Your answer is incorrect.

The correct answer is: It disables interrupts. It is required because the interrupt handlers of kernel are not yet installed.

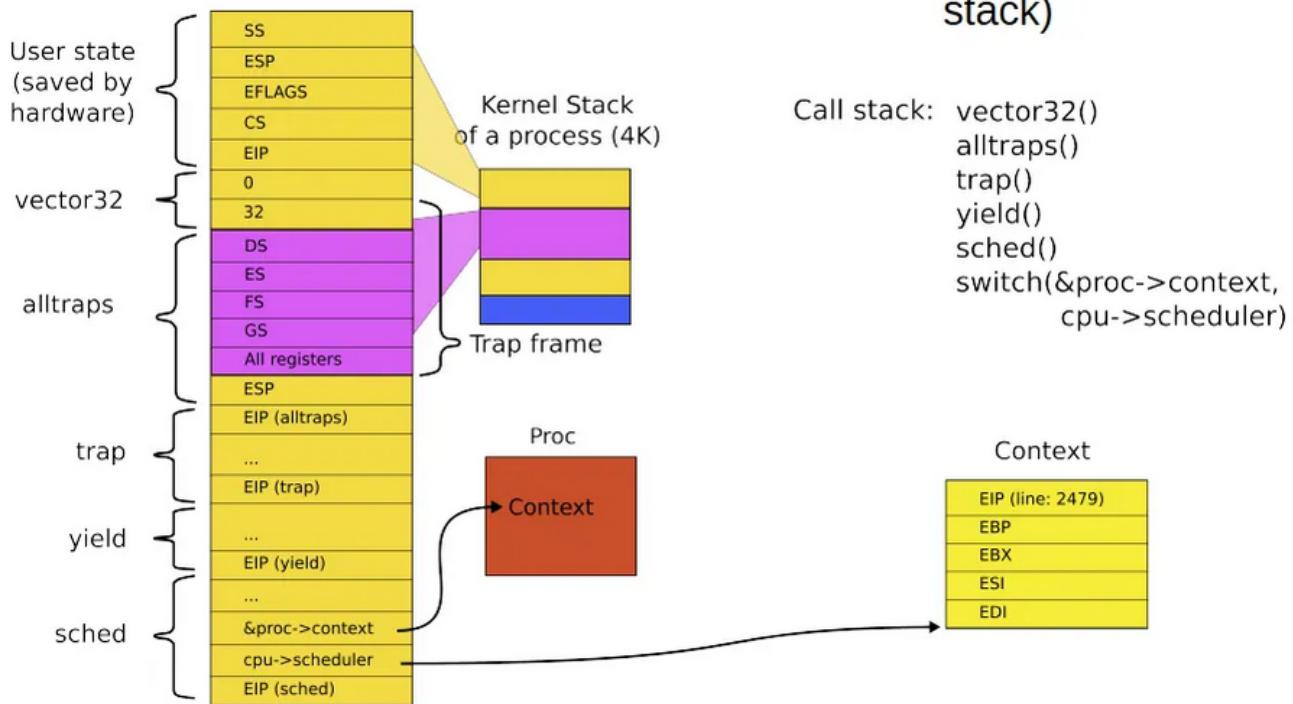
Question 13

Partially correct

Mark 0.42 out of 0.50

Mark statements as True/False, w.r.t. the given diagram

Stack inside swtch() and its two arguments (passed on the stack)



Call stack: vector32()
alltraps()
trap()
yield()
sched()
switch(&proc->context,
cpu->scheduler)

True False

<input type="radio"/> <input checked="" type="checkbox"/>	This is a diagram of swtch() called from scheduler()	<input checked="" type="checkbox"/> No. diagram of swtch() called from sched()
<input checked="" type="checkbox"/> <input type="radio"/> <input checked="" type="checkbox"/>	The "ESP" (second entry from top) is stack pointer of user-stack of process, while the "ESP" (first entry below pink region) is the trapframe pointer on kernel stack of process.	<input checked="" type="checkbox"/>
<input type="radio"/> <input checked="" type="checkbox"/>	The diagram is wrong because it shows the user stack and kernel stack together (continuous), but in practice they are separate	<input checked="" type="checkbox"/> diagram shows only kernel stack
<input checked="" type="checkbox"/> <input type="radio"/> <input checked="" type="checkbox"/>	The diagram is correct	<input checked="" type="checkbox"/>
<input checked="" type="checkbox"/> <input type="radio"/> <input checked="" type="checkbox"/>	The blue shaded part in "kernel stack of a process(4k)" refers to remaining part of stack (not used yet)	<input checked="" type="checkbox"/>
<input type="radio"/> <input checked="" type="checkbox"/>	The "context" yellow coloured box, pointed to by cpu->scheduler is on the kernel stack of the scheduler.	<input checked="" type="checkbox"/>

This is a diagram of swtch() called from scheduler(): False

The "ESP" (second entry from top) is stack pointer of user-stack of process, while the "ESP" (first entry below pink region) is the trapframe pointer on kernel stack of process.: True

The diagram is wrong because it shows the user stack and kernel stack together (continuous), but in practice they are separate: False

The diagram is correct: True

The blue shaded part in "kernel stack of a process(4k)" refers to remaining part of stack (not used yet): True
The "context" yellow coloured box, pointed to by cpu->scheduler is on the kernel stack of the scheduler.: False

Question 14

Partially correct

Mark 0.66 out of 0.75

Mark statements as True/False w.r.t. ptable.lock

True	False	
<input checked="" type="radio"/>	<input type="radio"/>	One sequence of function calls which takes and releases the ptable.lock is this: iderw->sleep, acquire(ptable.lock)->sched->swtch()->scheduler()->swtch()->yield(), release(ptable.lock)
<input checked="" type="radio"/>	<input type="radio"/>	It is taken by one process but released by another process, running on same processor
<input type="radio"/>	<input checked="" type="radio"/>	A process can sleep on ptable.lock if it can't acquire it.
<input checked="" type="radio"/>	<input type="radio"/>	ptable.lock protects the proc[] array and all struct proc in the array
<input type="radio"/>	<input checked="" type="radio"/>	ptable.lock is acquired but never released
<input type="radio"/>	<input checked="" type="radio"/>	The swtch() in scheduler() is called without holding the ptable.lock when control jumps to it from sched()
<input type="radio"/>	<input checked="" type="radio"/>	ptable.lock can be held by different processes on different processors at the same time
<input checked="" type="radio"/>	<input type="radio"/>	the rule of "never block holding a spinlock" does not apply to ptable.lock in xv6

One sequence of function calls which takes and releases the ptable.lock is this:
iderw->sleep, acquire(ptable.lock)->sched->swtch()->scheduler()->swtch()->yield(), release(ptable.lock): True
It is taken by one process but released by another process, running on same processor: True
A process can sleep on ptable.lock if it can't acquire it.: False
ptable.lock protects the proc[] array and all struct proc in the array: True
ptable.lock is acquired but never released: False
The swtch() in scheduler() is called without holding the ptable.lock when control jumps to it from sched(): False
ptable.lock can be held by different processes on different processors at the same time: False
the rule of "never block holding a spinlock" does not apply to ptable.lock in xv6: True

Question 15

Incorrect

Mark 0.00 out of 0.25

Why is there a call to knit2? Why is it not merged with knit1?

- a. call to seginit() makes it possible to actually use PHYSTOP in argument to knit2()
- b. Because there is a limit on the values that the arguments to knit1() can take.
- c. When knit1() is called there is a need for few page frames, but later knit2() is called to serve need of more page frames ✗
- d. knit2 refers to virtual addresses beyond 4MB, which are not mapped before kalloc() is called

The correct answer is: knit2 refers to virtual addresses beyond 4MB, which are not mapped before kalloc() is called

Question 16

Partially correct

Mark 0.54 out of 0.75

code line, MMU setting: Match the line of xv6 code with the MMU setup employed

movl \$(V2P_WO(entrypgdir)), %eax	protected mode with segmentation and 4 MB pages	✗
movw %ax, %gs	protected mode with only segmentation	✓
ljmp \$(SEG_KCODE<<3), \$start32	real mode	✓
inb \$0x64,%al	real mode	✓
readseg((uchar*)elf, 4096, 0);	protected mode with only segmentation	✓
orl \$CR0_PE, %eax	protected mode with segmentation and 4 MB pages	✗
jmp *%eax	protected mode with segmentation and 4 MB pages	✓

The correct answer is: movl \$(V2P_WO(entrypgdir)), %eax → protected mode with only segmentation, movw %ax, %gs → protected mode with only segmentation, ljmp \$(SEG_KCODE<<3), \$start32 → real mode, inb \$0x64,%al → real mode, readseg((uchar*)elf, 4096, 0); → protected mode with only segmentation, orl \$CR0_PE, %eax → real mode, jmp *%eax → protected mode with segmentation and 4 MB pages

Question 17

Incorrect

Mark 0.00 out of 0.50

We often use terms like "swtch() changes stack from process's kernel stack to scheduler's stack", or "the values are pushed on stack", or "the stack is initialized to the new page", etc. while discussing xv6 on x86.

Which of the following most accurately describes the meaning of "stack" in such sentences?

- a. the region of memory which is currently used as stack by processor X
- b. The stack segment
- c. The ss:esp pair
- d. The region of memory where the kernel remembers all the function calls made
- e. The "stack" variable declared in "stack.S" in xv6
- f. The stack variable used in the program being discussed
- g. The region of memory allocated by kernel for storing the parameters of functions

Your answer is incorrect.

The correct answer is: The ss:esp pair

Question 18

Incorrect

Mark 0.00 out of 0.25

Which of the following call sequence is impossible in xv6?

- a. Process 1: timer interrupt -> trap() -> yield() -> sched() -> switch() -> scheduler() -> Process 2 runs -> write -> sys_write() -> trap() -> ...
- b. Process 1: write() -> sys_write() -> file_write() -> writei() -> bread() -> bget() -> iderw() -> sleep() -> sched() -> switch() (jumps to) -> scheduler() -> swtch() (jumps to) -> Process 2 (return call sequence) sched() -> yield() -> trap -> user-code
- c. Process 1: write() -> sys_write() -> file_write() -- timer interrupt -> trap() -> yield() -> sched() -> switch() (jumps to) -> scheduler() -> swtch() X (jumps to) -> Process 2 (return call sequence) sched() -> yield() -> trap -> user-code

Your answer is incorrect.

The correct answer is: Process 1: timer interrupt -> trap() -> yield() -> sched() -> switch() -> scheduler() -> Process 2 runs -> write -> sys_write() -> trap() -> ...

Question 19

Correct

Mark 0.50 out of 0.50

Consider the following command and its output:

```
$ ls -lht xv6.img kernel
-rw-rw-r-- 1 abhijit abhijit 4.9M Feb 15 11:09 xv6.img
-rwxrwxr-x 1 abhijit abhijit 209K Feb 15 11:09 kernel*
```

Following code in bootmain()

```
readseg((uchar*)elf, 4096, 0);
```

and following selected lines from Makefile

```
xv6.img: bootblock kernel
dd if=/dev/zero of=xv6.img count=10000
dd if=bootblock of=xv6.img conv=notrunc
dd if=kernel of=xv6.img seek=1 conv=notrunc
```

```
kernel: $(OBJS) entry.o entryother initcode kernel.ld
$(LD) $(LDFLAGS) -T kernel.ld -o kernel entry.o $(OBJS) -b binary initcode entryother
$(OBJDUMP) -S kernel > kernel.asm
$(OBJDUMP) -t kernel | sed '1,/SYMBOL TABLE/d; s/ .* / /; /^$/d' > kernel.sym
```

Also read the code of bootmain() in xv6 kernel.

Select the options that describe the meaning of these lines and their correlation.

- a. readseg() reads first 4k bytes of kernel in memory ✓
- b. Although the size of the xv6.img file is ~5MB, only some part out of it is the bootloader+kernel code and remaining part is all zeroes. ✓
- c. The kernel disk image is ~5MB, the kernel within it is 209 kb, but bootmain() initially reads only first 4kb, and the later part is read using program headers in bootmain(). ✓
- d. Although the size of the kernel file is 209 Kb, only 4Kb out of it is the actual kernel code and remaining part is all zeroes.
- e. The kernel.ld file contains instructions to the linker to link the kernel properly ✓
- f. The kernel is compiled by linking multiple .o files created from .c files; and the entry.o, initcode, entryother files ✓
- g. The kernel.asm file is the final kernel file
- h. The bootmain() code does not read the kernel completely in memory
- i. The kernel disk image is ~5MB, the kernel within it is 209 kb, but bootmain() initially reads only first 4kb, and the later part is not read as it is user programs.

Your answer is correct.

The correct answers are: The kernel disk image is ~5MB, the kernel within it is 209 kb, but bootmain() initially reads only first 4kb, and the later part is read using program headers in bootmain(), readseg() reads first 4k bytes of kernel in memory, The kernel is compiled by linking multiple .o files created from .c files; and the entry.o, initcode, entryother files, The kernel.ld file contains instructions to the linker to link the kernel properly, Although the size of the xv6.img file is ~5MB, only some part out of it is the bootloader+kernel code and remaining part is all zeroes.

Question 20

Correct

Mark 0.50 out of 0.50

Mark statements as True/False w.r.t. the creation of free page list in xv6.

True	False	
<input checked="" type="radio"/>	<input type="radio"/>	the kmem.lock is used by kfree() and kalloc() only.
<input type="radio"/>	<input checked="" type="radio"/>	if(kmem.use_lock) acquire(&kmem.lock); this "if" condition is true, when kinit2() runs because multi-processor support has been enabled by now.
<input type="radio"/>	<input checked="" type="radio"/>	free page list is a singly circular linked list.
<input checked="" type="radio"/>	<input type="radio"/>	kmem.use_lock is set to 1 after free page list is created, so that kmem.lock is taken before accessing kmem.freelist.
<input checked="" type="radio"/>	<input type="radio"/>	The pointers that link the pages together are in the first 4 bytes of the pages themselves
<input checked="" type="radio"/>	<input type="radio"/>	if(kmem.use_lock) acquire(&kmem.lock); is not done when called from kinit1() because there is no need to take the lock when kinit1() is running because interrupts are disabled and only one processor is running

the kmem.lock is used by kfree() and kalloc() only.: True

if(kmem.use_lock)

acquire(&kmem.lock);

this "if" condition is true, when kinit2() runs because multi-processor support has been enabled by now.: False

free page list is a singly circular linked list.: False

kmem.use_lock is set to 1 after free page list is created, so that kmem.lock is taken before accessing kmem.freelist.: True

The pointers that link the pages together are in the first 4 bytes of the pages themselves: True

if(kmem.use_lock)

acquire(&kmem.lock);

is not done when called from kinit1() because there is no need to take the lock when kinit1() is running because interrupts are disabled and only one processor is running: True

[◀ Quiz-1\(24 Feb 2023\)](#)

Jump to...

[Pre-requisite Quiz \(old\) - use it for practice ►](#)

Started on Friday, 24 February 2023, 2:44 PM

State Finished

Completed on Friday, 24 February 2023, 4:26 PM

Time taken 1 hour 42 mins

Grade 9.53 out of 10.00 (95.27%)

Question 1

Correct

Mark 0.50 out of 0.50

Consider the two programs given below to implement the command (ignore the fact that error checks are not done on return values of functions)

```
$ ls . /tmp/asdfksdf >/tmp/ddd 2>&1
```

Program 1

```
int main(int argc, char *argv[]) {  
    int fd, n, i;  
    char buf[128];  
  
    fd = open("/tmp/ddd", O_WRONLY | O_CREAT, S_IRUSR | S_IWUSR);  
    close(1);  
    dup(fd);  
    close(2);  
    dup(fd);  
    execl("/bin/ls", "/bin/ls", ".", "/tmp/asldjfaldfs", NULL);  
}
```

Program 2

```
int main(int argc, char *argv[]) {  
    int fd, n, i;  
    char buf[128];  
  
    close(1);  
    fd = open("/tmp/ddd", O_WRONLY | O_CREAT, S_IRUSR | S_IWUSR);  
    close(2);  
    fd = open("/tmp/ddd", O_WRONLY | O_CREAT, S_IRUSR | S_IWUSR);  
    execl("/bin/ls", "/bin/ls", ".", "/tmp/asldjfaldfs", NULL);  
}
```

Select all the correct statements about the programs

Select one or more:

- a. Program 1 does 1>&2
- b. Only Program 1 is correct✓
- c. Program 1 makes sure that there is one file offset used for '2' and '1'✓
- d. Program 2 does 1>&2
- e. Both program 1 and 2 are incorrect
- f. Both programs are correct
- g. Program 1 ensures 2>&1 and does not ensure > /tmp/ddd
- h. Only Program 2 is correct
- i. Program 2 makes sure that there is one file offset used for '2' and '1'
- j. Program 1 is correct for > /tmp/ddd but not for 2>&1
- k. Program 2 ensures 2>&1 and does not ensure > /tmp/ddd
- l. Program 2 is correct for > /tmp/ddd but not for 2>&1

Your answer is correct.

The correct answers are: Only Program 1 is correct, Program 1 makes sure that there is one file offset used for '2' and '1'

Question 2

Partially correct

Mark 0.36 out of 0.50

You must have seen the error message "Segmentation fault, core dumped" very often.

With respect to this error message, mark the statements as True/False.

True	False	
<input type="radio"/> ✗	<input checked="" type="radio"/> ✗	On Linux, the message is printed only because the memory management scheme is segmentation
<input checked="" type="radio"/> ✗	<input type="radio"/> ✗	On Linux, the process was sent a SIGSEGV signal and the default handler for the signal is "Term", so the process is terminated.
<input type="radio"/> ✗	<input checked="" type="radio"/> ✗	The term "core" refers to the core code of the kernel.
<input type="radio"/> ✗	<input checked="" type="radio"/> ✗	The illegal memory access was detected by the kernel and the process was punished by kernel.
<input checked="" type="radio"/> ✗	<input type="radio"/> ✗	The image of the process is stored in a file called "core", if the ulimit allows so.
<input checked="" type="radio"/> ✗	<input type="radio"/> ✗	The core file can be analysed later using a debugger, to determine what went wrong.
<input checked="" type="radio"/> ✗	<input type="radio"/> ✗	The process has definitely performed illegal memory access.

On Linux, the message is printed only because the memory management scheme is segmentation.: False

On Linux, the process was sent a SIGSEGV signal and the default handler for the signal is "Term", so the process is terminated.: True

The term "core" refers to the core code of the kernel.: False

The illegal memory access was detected by the kernel and the process was punished by kernel.: False

The image of the process is stored in a file called "core", if the ulimit allows so.: True

The core file can be analysed later using a debugger, to determine what went wrong.: True

The process has definitely performed illegal memory access.: True

Question 3

Correct

Mark 0.50 out of 0.50

How does the distinction between kernel mode and user mode function as a rudimentary form of protection (security) ?

Select one:

- a. It prohibits invocation of kernel code completely, if a user program is running
- b. It prohibits a user mode process from running privileged instructions ✓
- c. It disallows hardware interrupts when a process is running
- d. It prohibits one process from accessing other process's memory

Your answer is correct.

The correct answer is: It prohibits a user mode process from running privileged instructions

Question 4

Correct

Mark 0.50 out of 0.50

Doing a lookup on the pathname /a/b/b/c/d for opening the file "d" requires reading ✓ no. of inodes. Assume that there are no hard/soft links on the path.

Write the answer as a number.

The correct answer is: 6

Question 5

Correct

Mark 0.50 out of 0.50

Select all the blocks that may need to be written back to disk (if updated, of-course), as "Yes", when an operation of deleting a file is carried out on ext2 file system.

An option has to be correct entirely to be marked "Yes"

One or more data bitmap blocks for the parent directory

 No ✓

Superblock

 Yes ✓

Possibly one block bitmap corresponding to the parent directory

 Yes ✓

One or multiple data blocks of the parent directory

 No ✓

Data blocks of the file

 No ✓

Block bitmap(s) for all the blocks of the file

 Yes ✓

Your answer is correct.

The correct answer is: One or more data bitmap blocks for the parent directory → No, Superblock → Yes, Possibly one block bitmap corresponding to the parent directory → Yes, One or multiple data blocks of the parent directory → No, Data blocks of the file → No, Block bitmap(s) for all the blocks of the file → Yes

Question 6

Correct

Mark 0.50 out of 0.50

Select the compiler's view of the process's address space, for each of the following MMU schemes:

(Assume that each scheme,e.g. paging/segmentation/etc is effectively utilised)

Relocation + Limit

 one continuous chunk ✓

Segmentation

 many continuous chunks of variable size ✓

Paging

 one continuous chunk ✓

Segmentation, then paging

 many continuous chunks of variable size ✓

Your answer is correct.

The correct answer is: Relocation + Limit → one continuous chunk, Segmentation → many continuous chunks of variable size, Paging → one continuous chunk, Segmentation, then paging → many continuous chunks of variable size

Question 7

Partially correct

Mark 0.36 out of 0.50

Mark the statements as True/False w.r.t. the basic concepts of memory management.

True	False	
<input type="radio"/> ✗	<input checked="" type="radio"/> ✗	The kernel refers to the page table for converting each virtual address to physical address.
<input checked="" type="radio"/> ✗	<input type="radio"/> ✗	When a process is executing, each virtual address is converted into physical address by the CPU hardware directly.
<input type="radio"/> ✗	<input checked="" type="radio"/> ✗	The compiler generates the address references for code/data/stack/heap in the executable file as per the memory management schema chosen by the compiler itself, and then the kernel ensures that program is executed with this schema.
<input type="radio"/> ✗	<input checked="" type="radio"/> ✗	The compiler interacts with the kernel continuously while compiling a program and obtains the correct set of memory addresses for code/stack/heap/data and then generates the machine code file.
<input checked="" type="radio"/> ✗	<input type="radio"/> ✗	The compiler generates address references for code/data/stack/heap in the executable file, depending on the MM architecture provided by CPU and kernel.
<input checked="" type="radio"/> ✗	<input type="radio"/> ✗	The kernel ensures that the MMU is setup before scheduling a process and then the CPU/MMU ensures that the address translation takes place.
<input type="radio"/> ✗	<input checked="" type="radio"/> ✗	When a process is executing, each virtual address is converted into physical address by the kernel directly.

The kernel refers to the page table for converting each virtual address to physical address.: False

When a process is executing, each virtual address is converted into physical address by the CPU hardware directly.: True

The compiler generates the address references for code/data/stack/heap in the executable file as per the memory management schema chosen by the compiler itself, and then the kernel ensures that program is executed with this schema.: False

The compiler interacts with the kernel continuously while compiling a program and obtains the correct set of memory addresses for code/stack/heap/data and then generates the machine code file.: False

The compiler generates address references for code/data/stack/heap in the executable file, depending on the MM architecture provided by CPU and kernel.: True

The kernel ensures that the MMU is setup before scheduling a process and then the CPU/MMU ensures that the address translation takes place.: True

When a process is executing, each virtual address is converted into physical address by the kernel directly.: False

Question 8

Correct

Mark 0.50 out of 0.50

Map each signal with its meaning

SIGCHLD	Child Stopped or Terminated	✓
SIGSEGV	Invalid Memory Reference	✓
SIGPIPE	Broken Pipe	✓
SIGUSR1	User Defined Signal	✓
SIGALRM	Timer Signal from alarm()	✓

The correct answer is: SIGCHLD → Child Stopped or Terminated, SIGSEGV → Invalid Memory Reference, SIGPIPE → Broken Pipe, SIGUSR1 → User Defined Signal, SIGALRM → Timer Signal from alarm()

Question 9

Partially correct

Mark 0.45 out of 0.50

Match the elements of C program to their place in memory

#include files	No memory needed	✓
Local Variables	Stack	✓
Arguments	Stack	✓
#define MACROS	No memory needed	✗
Code of main()	Code	✓
Function code	Code	✓
Local Static variables	Data	✓
Mallocoed Memory	Heap	✓
Global Static variables	Data	✓
Global variables	Data	✓

The correct answer is: #include files → No memory needed, Local Variables → Stack, Arguments → Stack, #define MACROS → No Memory needed, Code of main() → Code, Function code → Code, Local Static variables → Data, Mallocoed Memory → Heap, Global Static variables → Data, Global variables → Data

Question 10

Correct

Mark 0.50 out of 0.50

Predict the output of the program given here.

Assume that all the path names for the programs are correct. For example "/usr/bin/echo" will actually run echo command.

Assume that there is no mixing of printf output on screen if two of them run concurrently.

In the answer replace a new line by a single space.

For example::

good

output

should be written as good output

--

```
main() {  
    int i;  
    i = fork();  
    if(i == 0)  
        execl("/usr/bin/echo", "/usr/bin/echo", "hi", 0);  
    else  
        wait(0);  
    fork();  
    execl("/usr/bin/echo", "/usr/bin/echo", "one", 0);  
}
```

Answer: hi one one



The correct answer is: hi one one

Question 11

Correct

Mark 0.50 out of 0.50

Map the block allocation scheme with the problem it suffers from

(Match pairs 1-1, match a scheme with the problem that it suffers from relatively the most, compared to others)

Continuous allocation

need for compaction



Linked allocation

Too many seeks



Indexed Allocation

Overhead of reading metadata blocks



Your answer is correct.

The correct answer is: Continuous allocation → need for compaction, Linked allocation → Too many seeks, Indexed Allocation → Overhead of reading metadata blocks

Question 12

Partially correct

Mark 0.44 out of 0.50

How does the compiler calculate addresses for the different parts of a C program, when paging is used?

Global variables	Immediately after the text	✓
Static variables	Immediately after the text, along with globals	✓
#include files	No memory allocated, they are handled by linker	✗
malloced memory	Heap (handled by the malloc-free library, using OS's system calls)	✓
typedef	No memory allocated, as they are not variables, but only conceptual definition of a type	✓
#define	No memory allocated, they are handled by pre-processor	✓
Local variables	An offset with respect to stack pointer (esp)	✓
Text	starting with 0	✓

Your answer is partially correct.

You have correctly selected 7.

The correct answer is: Global variables → Immediately after the text, Static variables → Immediately after the text, along with globals, #include files → No memory allocated for the file, but if it contains variables, then variables may be allocated memory, malloced memory → Heap (handled by the malloc-free library, using OS's system calls), typedef → No memory allocated, as they are not variables, but only conceptual definition of a type, #define → No memory allocated, they are handled by pre-processor, Local variables → An offset with respect to stack pointer (esp), Text → starting with 0

Question 13

Correct

Mark 0.50 out of 0.50

Mark the statements about named and un-named pipes as True or False

True	False	
<input type="radio"/> ✗	<input checked="" type="checkbox"/> ✗	A named pipe has a name decided by the kernel.
<input checked="" type="checkbox"/> ✗	<input type="radio"/> ✗	Named pipes can exist beyond the life-time of processes using them.
<input checked="" type="checkbox"/> ✗	<input type="radio"/> ✗	Un-named pipes can be used for communication between only "related" processes, if the common ancestor created it.
<input type="radio"/> ✗	<input checked="" type="checkbox"/> ✗	The buffers for named-pipe are in process-memory while the buffers for the un-named pipe are in kernel memory.
<input checked="" type="checkbox"/> ✗	<input type="radio"/> ✗	Both types of pipes provide FIFO communication.
<input type="radio"/> ✗	<input checked="" type="checkbox"/> ✗	The pipe() system call can be used to create either a named or un-named pipe.
<input type="radio"/> ✗	<input checked="" type="checkbox"/> ✗	Named pipes can be used for communication between only "related" processes.
<input checked="" type="checkbox"/> ✗	<input type="radio"/> ✗	Un-named pipes are inherited by a child process from parent.
<input checked="" type="checkbox"/> ✗	<input type="radio"/> ✗	Both types of pipes are an extension of the idea of "message passing".
<input checked="" type="checkbox"/> ✗	<input type="radio"/> ✗	Named pipe exists as a file

A named pipe has a name decided by the kernel.: False

Named pipes can exist beyond the life-time of processes using them.: True

Un-named pipes can be used for communication between only "related" processes, if the common ancestor created it.: True

The buffers for named-pipe are in process-memory while the buffers for the un-named pipe are in kernel memory.: False

Both types of pipes provide FIFO communication.: True

The pipe() system call can be used to create either a named or un-named pipe.: False

Named pipes can be used for communication between only "related" processes.: False

Un-named pipes are inherited by a child process from parent.: True

Both types of pipes are an extension of the idea of "message passing": True

Named pipe exists as a file: True

Question 14

Partially correct

Mark 0.45 out of 0.50

Select Yes if the mentioned element should be a part of PCB

Select No otherwise.

Yes	No	
<input checked="" type="radio"/>	<input type="radio"/> <input checked="" type="checkbox"/>	List of opened files
<input checked="" type="radio"/>	<input type="radio"/> <input checked="" type="checkbox"/>	EIP at the time of context switch
<input type="radio"/> <input checked="" type="checkbox"/>	<input checked="" type="radio"/>	Pointer to IDT
<input checked="" type="radio"/>	<input checked="" type="radio"/> <input type="checkbox"/>	Pointer to the parent process
<input checked="" type="radio"/>	<input type="radio"/> <input checked="" type="checkbox"/>	PID
<input checked="" type="radio"/>	<input type="radio"/> <input checked="" type="checkbox"/>	Memory management information about that process
<input checked="" type="radio"/>	<input type="radio"/> <input checked="" type="checkbox"/>	Process context
<input checked="" type="radio"/>	<input type="radio"/> <input checked="" type="checkbox"/>	Process state
<input type="radio"/> <input checked="" type="checkbox"/>	<input checked="" type="radio"/>	PID of Init
<input type="radio"/> <input checked="" type="checkbox"/>	<input checked="" type="radio"/>	Function pointers to all system calls

List of opened files: Yes

EIP at the time of context switch: Yes

Pointer to IDT: No

Pointer to the parent process: Yes

PID: Yes

Memory management information about that process: Yes

Process context: Yes

Process state: Yes

PID of Init: No

Function pointers to all system calls: No

Question 15

Correct

Mark 0.50 out of 0.50

Which of the following parts of a C program do not have any corresponding machine code ?

- a. #directives✓
- b. function calls
- c. local variable declaration
- d. pointer dereference
- e. global variables✓
- f. typedefs✓
- g. expressions

Your answer is correct.

The correct answers are: #directives, typedefs, global variables

Question 16

Correct

Mark 0.50 out of 0.50

Mark the statements about device drivers by marking as True or False.

True False

<input checked="" type="radio"/>	<input type="radio"/> X	Device driver is an intermediary between the hardware controller and OS	✓
<input type="radio"/> X	<input checked="" type="radio"/>	Device driver is part of hardware	✓
<input checked="" type="radio"/>	<input type="radio"/> X	Device driver is part of OS code	✓
<input type="radio"/> X	<input checked="" type="radio"/>	Different devices of the same type (e.g. 2 IDE hard disks) must need different device drivers.	✓
<input checked="" type="radio"/>	<input type="radio"/> X	It's possible that a particular hardware has multiple device drivers available for it.	✓
<input checked="" type="radio"/>	<input type="radio"/> X	Writing a device driver mandatorily demands reading the technical documentation about the hardware.	✓

Device driver is an intermediary between the hardware controller and OS.: True

Device driver is part of hardware: False

Device driver is part of OS code: True

Different devices of the same type (e.g. 2 IDE hard disks) must need different device drivers.: False

It's possible that a particular hardware has multiple device drivers available for it.: True

Writing a device driver mandatorily demands reading the technical documentation about the hardware.: True

Question 17

Partially correct

Mark 0.48 out of 0.50

Following code claims to implement the command

/bin/ls -l | /usr/bin/head -3 | /usr/bin/tail -1

Fill in the blanks to make the code work.

Note: Do not include space in writing any option. x[1][2] should be written without any space, and so is the case with [1] or [2]. Pay attention to exact syntax and do not write any extra character like ';' or '=' etc.

```
int main(int argc, char *argv[]) {
```

```
    int pid1, pid2;
```

```
    int pfd[
```

```
    2
```

```
    ][2];
```

```
    pipe(
```

```
    pfd[0]
```

```
    );
```

```
    pid1 =
```

```
    fork()
```

```
    ;
```

```
    if(pid1 != 0) {
```

```
        close(pfd[0]
```

```
        [0]
```

```
    );
```

```
    close(
```

```
    1
```

```
    );
```

```
    dup(
```

```
    pfd[0][1]
```

```
    );
```

```
    execl("/bin/ls", "/bin/ls", "
```

```
    -l
```

```
    ", NULL);
```

```
    }
```

```
    pipe(
```

```
    pfd[1]
```

```
    );
```

```
    pid2
```

```
    = fork();
```

```
    if(pid2 == 0) {
```

```
        close(
```

```
        pfd[0][1] )
```

```
    ;
```

```
    close(0);
```

```
    dup(
```

```
    pfd[0][0]
```

```

✓ );
close(pfd[1]
[0]

✓ );
close(
1

✓ );
dup(
pfd[1][1]

✓ );
execl("/usr/bin/head", "/usr/bin/head", "
-3

✓ ", NULL);
} else {
close(pfd
[1][1]

✓ );
close(
0

✓ );
dup(
pfd[1][0]

✓ );
close(pfd
[0][0]

✓ );
execl("/usr/bin/tail", "/usr/bin/tail", "
-1

✓ ", NULL);
}
}

```

Question 18

Correct

Mark 0.50 out of 0.50

What is meant by formatting a disk/partition?

- a. storing all the necessary programs on the disk/partition
- b. erasing all data on the disk/partition
- c. writing zeroes on all sectors
- d. creating layout of empty directory tree/graph data structure✓

The correct answer is: creating layout of empty directory tree/graph data structure

Question 19

Correct

Mark 0.50 out of 0.50

Which of the following instructions should be privileged?

Select one or more:

- a. Read the clock.
- b. Access memory management unit of the processor ✓
- c. Set value of a memory location
- d. Turn off interrupts. ✓
- e. Set value of timer. ✓
- f. Access a general purpose register
- g. Access I/O device. ✓
- h. Switch from user to kernel mode. ✓ This instruction (like INT) is itself privileged - and that is why it not only changes the mode, but also ensures a jump to an ISR (kernel code)
- i. Modify entries in device-status table ✓

Your answer is correct.

The correct answers are: Set value of timer., Access memory management unit of the processor, Turn off interrupts., Modify entries in device-status table, Access I/O device., Switch from user to kernel mode.

Question 20

Correct

Mark 0.50 out of 0.50

Mark statements True/False w.r.t. change of states of a process. Note that a statement is true only if the claim and argument both are true.

Reference: The process state diagram (and your understanding of how kernel code works). Note - the diagram does not show zombie state!

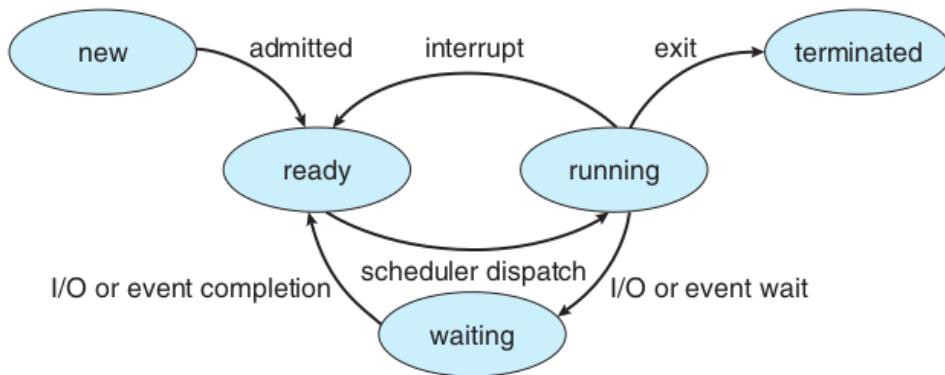


Figure 3.2 Diagram of process state.

True	False
<input checked="" type="radio"/>	<input type="radio"/> ✗
<input type="radio"/> ✗	<input checked="" type="radio"/>
<input type="radio"/> ✗	<input checked="" type="radio"/>
<input type="radio"/> ✗	<input checked="" type="radio"/>
<input checked="" type="radio"/>	<input type="radio"/> ✗
<input checked="" type="radio"/>	<input type="radio"/> ✗

Only a process in READY state is considered by scheduler



A process only in RUNNING state can become TERMINATED because scheduler moves it to ZOMBIE state first



A process in READY state can not go to WAITING state because the resource on which it will WAIT will not be in use when process is in READY state.



Every forked process has to go through ZOMBIE state, at least for a small duration.



A process in WAITING state can not become RUNNING because the event it's waiting for has not occurred and it has not been moved to ready queue yet



Only a process in READY state is considered by scheduler: True

A process only in RUNNING state can become TERMINATED because scheduler moves it to ZOMBIE state first: False

A process in READY state can not go to WAITING state because the resource on which it will WAIT will not be in use when process is in READY state.: False

Every forked process has to go through ZOMBIE state, at least for a small duration.: True

A process in WAITING state can not become RUNNING because the event it's waiting for has not occurred and it has not been moved to ready queue yet: True

[◀ Quiz-1 Preparation questions](#)

Jump to...

Started on Sunday, 18 February 2024, 2:00 PM

State Finished

Completed on Sunday, 18 February 2024, 3:56 PM

Time taken 1 hour 55 mins

Grade 13.45 out of 15.00 (89.69%)

Question 1

Correct

Mark 1.00 out of 1.00

Match the program with its output (ignore newlines in the output. Just focus on the count of 'hi')

main() { fork(); execl("/usr/bin/echo", "/usr/bin/echo", "hi\n", NULL); }
main() { execl("/usr/bin/echo", "/usr/bin/echo", "hi\n", NULL); }
main() { int i = NULL; fork(); printf("hi\n"); }
main() { int i = fork(); if(i == 0) execl("/usr/bin/echo", "/usr/bin/echo", "hi\n", NULL); }

- hi hi ✓
- hi ✓
- hi hi ✓
- hi ✓

Your answer is correct.

The correct answer is: main() { fork(); execl("/usr/bin/echo", "/usr/bin/echo", "hi\n", NULL); } → hi hi, main() { execl("/usr/bin/echo", "/usr/bin/echo", "hi\n", NULL); } → hi, main() { int i = NULL; fork(); printf("hi\n"); } → hi hi, main() { int i = fork(); if(i == 0) execl("/usr/bin/echo", "/usr/bin/echo", "hi\n", NULL); } → hi

Question 2

Partially correct

Mark 0.40 out of 0.50

Match the elements of C program to their place in memory

Global Static variables	Data	✓
Local Variables	Stack	✓
#define MACROS	No Memory needed	✓
Code of main()	Data	✗
#include files	No Memory needed	✗
Global variables	Data	✓
Function code	Code	✓
Mallocoed Memory	Heap	✓
Arguments	Stack	✓
Local Static variables	Data	✓

The correct answer is: Global Static variables → Data, Local Variables → Stack, #define MACROS → No Memory needed, Code of main() → Code, #include files → No memory needed, Global variables → Data, Function code → Code, Mallocoed Memory → Heap, Arguments → Stack, Local Static variables → Data

Question 3

Correct

Mark 1.00 out of 1.00

Consider the image given below, which explains how paging works.

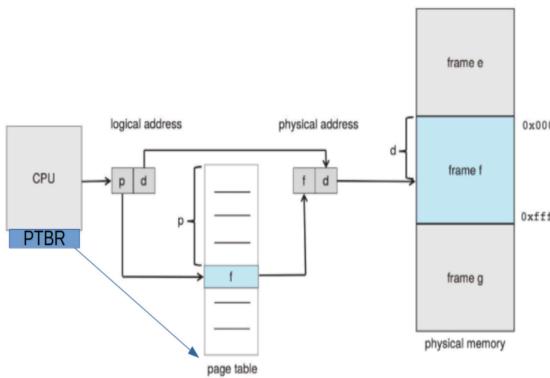


Figure 9.8 Paging hardware.

Mention whether each statement is True or False, with respect to this image.

True	False	
<input type="radio"/> ✗	<input checked="" type="radio"/> ✗	The page table is indexed using frame number
<input type="radio"/> ✗	<input checked="" type="radio"/> ✗	Size of page table is always determined by the size of RAM
<input checked="" type="radio"/> ✗	<input type="radio"/> ✗	The PTBR is present in the CPU as a register
<input type="radio"/> ✗	<input checked="" type="radio"/> ✗	The locating of the page table using PTBR also involves paging translation
<input checked="" type="radio"/> ✗	<input type="radio"/> ✗	The page table is itself present in Physical memory
<input checked="" type="radio"/> ✗	<input type="radio"/> ✗	The page table is indexed using page number
<input checked="" type="radio"/> ✗	<input type="radio"/> ✗	The physical address may not be of the same size (in bits) as the logical address
<input checked="" type="radio"/> ✗	<input type="radio"/> ✗	Maximum Size of page table is determined by number of bits used for page number

The page table is indexed using frame number: False

Size of page table is always determined by the size of RAM: False

The PTBR is present in the CPU as a register: True

The locating of the page table using PTBR also involves paging translation: False

The page table is itself present in Physical memory: True

The page table is indexed using page number: True

The physical address may not be of the same size (in bits) as the logical address: True

Maximum Size of page table is determined by number of bits used for page number: True

Question 4

Partially correct

Mark 0.75 out of 1.00

Select all the correct statements about calling convention on x86 32-bit.

- a. Parameters are pushed on the stack in left-right order
- b. Space for local variables is allocated by subtracting the stack pointer inside the code of the called function ✓
- c. Compiler may allocate more memory on stack than needed ✓
- d. Parameters may be passed in registers or on stack ✓
- e. during execution of a function, ebp is pointing to the old ebp ✓
- f. Return address is one location above the ebp ✓
- g. The ebp pointers saved on the stack constitute a chain of activation records ✓
- h. The return value is either stored on the stack or returned in the eax register
- i. The two lines in the beginning of each function, "push %ebp; mov %esp, %ebp", create space for local variables
- j. Space for local variables is allocated by subtracting the stack pointer inside the code of the caller function ✗
- k. Parameters may be passed in registers or on stack ✓

Your answer is partially correct.

You have selected too many options.

The correct answers are: Compiler may allocate more memory on stack than needed, Parameters may be passed in registers or on stack, Parameters may be passed in registers or on stack, Return address is one location above the ebp, during execution of a function, ebp is pointing to the old ebp, Space for local variables is allocated by subtracting the stack pointer inside the code of the called function, The ebp pointers saved on the stack constitute a chain of activation records

Question **5**

Correct

Mark 0.50 out of 0.50

What's the trapframe in xv6?

- a. A frame of memory that contains all the trap handler code's function pointers
- b. The sequence of values, including saved registers, constructed on the stack when an interrupt occurs, built by hardware only
- c. The IDT table
- d. The sequence of values, including saved registers, constructed on the stack when an interrupt occurs, built by hardware + code ✓
in trapasm.S
- e. A frame of memory that contains all the trap handler code
- f. The sequence of values, including saved registers, constructed on the stack when an interrupt occurs, built by code in trapasm.S only
- g. A frame of memory that contains all the trap handler's addresses

Your answer is correct.

The correct answer is: The sequence of values, including saved registers, constructed on the stack when an interrupt occurs, built by hardware + code in trapasm.S

Question **6**

Correct

Mark 0.50 out of 0.50

Match the File descriptors to their meaning

- | | | |
|---|-----------------|---|
| 1 | Standard output | ✓ |
| 2 | Standard error | ✓ |
| 0 | Standard Input | ✓ |

The correct answer is: 1 → Standard output, 2 → Standard error, 0 → Standard Input

Question **7**

Incorrect

Mark 0.00 out of 0.50

The ljmp instruction in general does

- a. change the CS and EIP to 32 bit mode, and jumps to next line of code **X**
- b. change the CS and EIP to 32 bit mode
- c. change the CS and EIP to 32 bit mode, and jumps to new value of EIP
- d. change the CS and EIP to 32 bit mode, and jumps to kernel code

The correct answer is: change the CS and EIP to 32 bit mode, and jumps to new value of EIP

Question 8

Correct

Mark 1.00 out of 1.00

Mark the statements as True/False w.r.t. the basic concepts of memory management.

True False

<input type="radio"/> ✗	<input checked="" type="radio"/> ✗	The kernel refers to the page table for converting each virtual address to physical address.	✓
<input checked="" type="radio"/> ✗	<input type="radio"/> ✗	The compiler generates address references for code/data/stack/heap in the executable file, depending on the MM architecture provided by CPU and kernel.	✓
<input type="radio"/> ✗	<input checked="" type="radio"/> ✗	The compiler generates the address references for code/data/stack/heap in the executable file as per the memory management schema chosen by the compiler itself, and then the kernel ensures that program is executed with this schema.	✓
<input type="radio"/> ✗	<input checked="" type="radio"/> ✗	When a process is executing, each virtual address is converted into physical address by the kernel directly.	✓
<input checked="" type="radio"/> ✗	<input type="radio"/> ✗	The kernel ensures that the MMU is setup before scheduling a process and then the CPU/MMU ensures that the address translation takes place.	✓
<input type="radio"/> ✗	<input checked="" type="radio"/> ✗	The compiler interacts with the kernel continuously while compiling a program and obtains the correct set of memory addresses for code/stack/heap/data and then generates the machine code file.	✓
<input checked="" type="radio"/> ✗	<input type="radio"/> ✗	When a process is executing, each virtual address is converted into physical address by the CPU hardware directly.	✓

The kernel refers to the page table for converting each virtual address to physical address.: False

The compiler generates address references for code/data/stack/heap in the executable file, depending on the MM architecture provided by CPU and kernel.: True

The compiler generates the address references for code/data/stack/heap in the executable file as per the memory management schema chosen by the compiler itself, and then the kernel ensures that program is executed with this schema.: False

When a process is executing, each virtual address is converted into physical address by the kernel directly.: False

The kernel ensures that the MMU is setup before scheduling a process and then the CPU/MMU ensures that the address translation takes place.: True

The compiler interacts with the kernel continuously while compiling a program and obtains the correct set of memory addresses for code/stack/heap/data and then generates the machine code file.: False

When a process is executing, each virtual address is converted into physical address by the CPU hardware directly.: True

Question 9

Correct

Mark 1.00 out of 1.00

Suppose a processor supports base(relocation register) + limit scheme of MMU.

Assuming this, mark the statements as True/False

True	False	
<input checked="" type="radio"/>	<input type="radio"/> X	The OS sets up the relocation and limit registers when the process is scheduled
<input checked="" type="radio"/>	<input type="radio"/> X	The compiler generates machine code assuming continuous memory address space for process, and calculating appropriate sizes for code, and data;
<input type="radio"/> X	<input checked="" type="radio"/>	The OS detects any memory access beyond the limit value and raises an interrupt
<input type="radio"/> X	<input checked="" type="radio"/>	The compiler generates machine code assuming appropriately sized semgments for code, data and stack.
<input type="radio"/> X	<input checked="" type="radio"/>	The process sets up it's own relocation and limit registers when the process is scheduled
<input checked="" type="radio"/>	<input type="radio"/> X	The hardware detects any memory access beyond the limit value and raises an interrupt
<input checked="" type="radio"/>	<input type="radio"/> X	The OS may terminate the process while handling the interrupt of memory violation
<input type="radio"/> X	<input checked="" type="radio"/>	The hardware may terminate the process while handling the interrupt of memory violation

The OS sets up the relocation and limit registers when the process is scheduled: True

The compiler generates machine code assuming continuous memory address space for process, and calculating appropriate sizes for code, and data;: True

The OS detects any memory access beyond the limit value and raises an interrupt: False

The compiler generates machine code assuming appropriately sized semgments for code, data and stack.: False

The process sets up it's own relocation and limit registers when the process is scheduled: False

The hardware detects any memory access beyond the limit value and raises an interrupt: True

The OS may terminate the process while handling the interrupt of memory violation: True

The hardware may terminate the process while handling the interrupt of memory violation: False

Question 10

Correct

Mark 0.50 out of 0.50

Match the register pairs

IP	CS	✓
BP	SS	✓
DI	DS	✓
SI	DS	✓
SP	SS	✓

The correct answer is: IP → CS, BP → SS, DI → DS, SI → DS, SP → SS

Question 11

Correct

Mark 0.50 out of 0.50

Some part of the bootloader of xv6 is written in assembly while some part is written in C. Why is that so?

Select all the appropriate choices

- a. The code for reading ELF file can not be written in assembly
- b. The code in assembly is required for transition to protected mode, from real mode; but calling convention was applicable all the time
- c. The setting up of the most essential memory management infrastructure needs assembly code ✓
- d. The code in assembly is required for transition to protected mode, from real mode; after that calling convention applies, hence code can be written in C ✓

Your answer is correct.

The correct answers are: The code in assembly is required for transition to protected mode, from real mode; after that calling convention applies, hence code can be written in C, The setting up of the most essential memory management infrastructure needs assembly code

Question **12**

Correct

Mark 0.50 out of 0.50

A process blocks itself means

- a. The kernel code of an interrupt handler, moves the process to a waiting queue and calls scheduler
- b. The application code calls the scheduler
- c. The kernel code of system call, called by the process, moves the process to a waiting queue and calls scheduler ✓
- d. The kernel code of system call calls scheduler

The correct answer is: The kernel code of system call, called by the process, moves the process to a waiting queue and calls scheduler

Question **13**

Correct

Mark 0.50 out of 0.50

The kernel is loaded at Physical Address

- a. 0x80000000
- b. 0x00100000 ✓
- c. 0x80100000
- d. 0x0010000

The correct answer is: 0x00100000

Question 14

Correct

Mark 0.50 out of 0.50

```
int value = 5;
int main()
{
    pid_t pid;
    pid = fork();
    if (pid == 0) { /* child process */
        value += 15;
        return 0;
    }
    else if (pid > 0) { /* parent process */
        wait(NULL);
        printf("%d", value); /* LINE A */
    }
    return 0;
}
```

What's the value printed here at LINE A?

Answer: 5



The correct answer is: 5

Question 15

Incorrect

Mark 0.00 out of 0.50

The variable \$stack in entry.S is

- a. a memory region allocated as a part of entry.S
- b. located at less than 0x7c00
- c. located at 0x7c00
- d. located at 0
- e. located at the value given by %esp as setup by bootmain()

The correct answer is: a memory region allocated as a part of entry.S

Question **16**

Correct

Mark 0.50 out of 0.50

The right side of line of code "entry = (void(*)(void))(elf->entry)" means

- a. Get the "entry" in ELF structure and convert it into a function void pointer
- b. Convert the "entry" in ELF structure into void
- c. Get the "entry" in ELF structure and convert it into a void pointer
- d. Get the "entry" in ELF structure and convert it into a function pointer accepting no arguments and returning nothing ✓

The correct answer is: Get the "entry" in ELF structure and convert it into a function pointer accepting no arguments and returning nothing ✓

Question 17

Partially correct

Mark 0.30 out of 0.50

xv6.img: bootblock kernel

```
dd if=/dev/zero of=xv6.img count=10000
dd if=bootblock of=xv6.img conv=notrunc
dd if=kernel of=xv6.img seek=1 conv=notrunc
```

Consider above lines from the Makefile. Which of the following is INCORRECT?

- a. xv6.img is the virtual processor used by the qemu emulator ✓
- b. The size of the kernel file is nearly 5 MB
- c. Blocks in xv6.img after kernel may be all zeroes.
- d. The size of xv6.img is exactly = (size of bootblock) + (size of kernel) ✓
- e. The size of the xv6.img is nearly 5 MB ✗
- f. The bootblock may be 512 bytes or less (looking at the Makefile instruction)
- g. The xv6.img is of the size 10,000 blocks of 512 bytes each and occupies 10,000 blocks on the disk.
- h. The kernel is located at block-1 of the xv6.img
- i. The xv6.img is the virtual disk that is created by combining the bootblock and the kernel file.
- j. The bootblock is located on block-0 of the xv6.img
- k. The xv6.img is of the size 10,000 blocks of 512 bytes each and occupies upto 10,000 blocks on the disk. ✓

Your answer is partially correct.

You have correctly selected 3.

The correct answers are: xv6.img is the virtual processor used by the qemu emulator, The xv6.img is of the size 10,000 blocks of 512 bytes each and occupies upto 10,000 blocks on the disk., The size of the kernel file is nearly 5 MB, The size of xv6.img is exactly = (size of bootblock) + (size of kernel)

Question 18

Correct

Mark 0.50 out of 0.50

How does the distinction between kernel mode and user mode function as a rudimentary form of protection (security) ?

Select one:

- a. It disallows hardware interrupts when a process is running
- b. It prohibits one process from accessing other process's memory
- c. It prohibits a user mode process from running privileged instructions ✓
- d. It prohibits invocation of kernel code completely, if a user program is running

Your answer is correct.

The correct answer is: It prohibits a user mode process from running privileged instructions

Question 19

Correct

Mark 1.00 out of 1.00

Select all the correct statements about MMU and it's functionality (on a non-demand paged system)

Select one or more:

- a. The Operating system sets up relevant CPU registers to enable proper MMU translations ✓
- b. The operating system interacts with MMU for every single address translation
- c. Illegal memory access is detected by operating system
- d. MMU is a separate chip outside the processor
- e. MMU is inside the processor ✓
- f. Logical to physical address translations in MMU are done with specific machine instructions
- g. Illegal memory access is detected in hardware by MMU and a trap is raised ✓
- h. Logical to physical address translations in MMU are done in hardware, automatically ✓

Your answer is correct.

The correct answers are: MMU is inside the processor, Logical to physical address translations in MMU are done in hardware, automatically, The Operating system sets up relevant CPU registers to enable proper MMU translations, Illegal memory access is detected in hardware by MMU and a trap is raised

Question **20**

Correct

Mark 0.50 out of 0.50

The trapframe, in xv6, is built by the

- a. hardware, trapasm.S
- b. hardware, vectors.S, trapasm.S ✓
- c. hardware, vectors.S, trapasm.S, trap()
- d. hardware, vectors.S
- e. vectors.S, trapasm.S

The correct answer is: hardware, vectors.S, trapasm.S

Question **21**

Correct

Mark 0.50 out of 0.50

The variable 'end' used as argument to kinit1 has the value

- a. 80102da0
- b. 8010a48c
- c. 81000000
- d. 801154a8 ✓
- e. 80110000
- f. 80000000

The correct answer is: 801154a8

Question **22**

Correct

Mark 0.50 out of 0.50

Select all the correct statements about zombie processes

Select one or more:

- a. If the parent of a process finishes, before the process itself, then after finishing the process is typically attached to 'init' as parent✓
- b. A process can become zombie if it finishes, but the parent has finished before it✓
- c. init() typically keeps calling wait() for zombie processes to get cleaned up✓
- d. A zombie process remains zombie forever, as there is no way to clean it up
- e. A process becomes zombie when its parent finishes
- f. A process becomes zombie when it finishes, and remains zombie until parent calls wait() on it✓
- g. A zombie process occupies space in OS data structures✓
- h. Zombie processes are harmless even if OS is up for long time

Your answer is correct.

The correct answers are: A process becomes zombie when it finishes, and remains zombie until parent calls wait() on it, A process can become zombie if it finishes, but the parent has finished before it, A zombie process occupies space in OS data structures, If the parent of a process finishes, before the process itself, then after finishing the process is typically attached to 'init' as parent, init() typically keeps calling wait() for zombie processes to get cleaned up

Question **23**

Correct

Mark 0.50 out of 0.50

Which of the following state transitions are not possible?

- a. Waiting -> Terminated✓
- b. Ready -> Terminated✓
- c. Ready -> Waiting✓
- d. Running -> Waiting

The correct answers are: Ready -> Terminated, Waiting -> Terminated, Ready -> Waiting

Question **24**

Correct

Mark 0.50 out of 0.50

The number of GDT entries setup during boot process of xv6 is

- a. 0
- b. 256
- c. 255
- d. 3 ✓
- e. 2
- f. 4

The correct answer is: 3

[◀ Homework questions: Basics of MM, xv6 booting](#)

Jump to...

[Quiz-2 \(15 Marks\) ►](#)

Started on Saturday, 16 March 2024, 1:32 PM

State Finished

Completed on Saturday, 16 March 2024, 3:33 PM

Time taken 2 hours 1 min

Grade 13.35 out of 15.00 (88.97%)

Question 1

Correct

Mark 0.50 out of 0.50

Select the correct statements about sched() and scheduler() in xv6 code

- a. Each call to sched() or scheduler() involves change of one stack inside swtch() ✓
- b. When either sched() or scheduler() is called, it does not return immediately to caller ✓
- c. sched() switches to the scheduler's context ✓
- d. sched() and scheduler() are co-routines ✓
- e. When either sched() or scheduler() is called, it results in a context switch ✓
- f. After call to swtch() in scheduler(), the control moves to code in sched() ✓
- g. scheduler() switches to the selected process's context ✓
- h. After call to swtch() in sched(), the control moves to code in scheduler() ✓

Your answer is correct.

The correct answers are: sched() and scheduler() are co-routines, When either sched() or scheduler() is called, it does not return immediately to caller, When either sched() or scheduler() is called, it results in a context switch, sched() switches to the scheduler's context, scheduler() switches to the selected process's context, After call to swtch() in scheduler(), the control moves to code in sched(), After call to swtch() in sched(), the control moves to code in scheduler(), Each call to sched() or scheduler() involves change of one stack inside swtch()

Question 2

Correct

Mark 0.50 out of 0.50

The variable \$stack in entry.S is

- a. located at 0x7c00
- b. located at the value given by %esp as setup by bootmain()
- c. a memory region allocated as a part of entry.S ✓
- d. located at less than 0x7c00
- e. located at 0

The correct answer is: a memory region allocated as a part of entry.S

Question 3

Correct

Mark 0.50 out of 0.50

Consider the image given below, which explains how paging works.

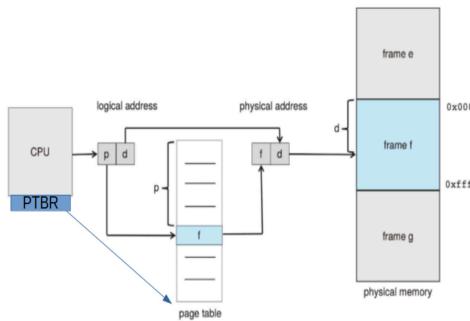


Figure 9.8 Paging hardware.

Mention whether each statement is True or False, with respect to this image.

True	False	
<input type="radio"/> <input checked="" type="checkbox"/>	<input checked="" type="radio"/> <input type="checkbox"/>	Size of page table is always determined by the size of RAM
<input checked="" type="checkbox"/>	<input type="radio"/> <input checked="" type="checkbox"/>	The PTBR is present in the CPU as a register
<input checked="" type="checkbox"/>	<input type="radio"/> <input checked="" type="checkbox"/>	Maximum Size of page table is determined by number of bits used for page number
<input type="radio"/> <input checked="" type="checkbox"/>	<input checked="" type="radio"/> <input type="checkbox"/>	The page table is indexed using frame number
<input checked="" type="checkbox"/>	<input type="radio"/> <input checked="" type="checkbox"/>	The page table is itself present in Physical memory
<input checked="" type="checkbox"/>	<input type="radio"/> <input checked="" type="checkbox"/>	The page table is indexed using page number
<input checked="" type="checkbox"/>	<input type="radio"/> <input checked="" type="checkbox"/>	The physical address may not be of the same size (in bits) as the logical address
<input type="radio"/> <input checked="" type="checkbox"/>	<input checked="" type="radio"/> <input type="checkbox"/>	The locating of the page table using PTBR also involves paging translation

Size of page table is always determined by the size of RAM: False

The PTBR is present in the CPU as a register: True

Maximum Size of page table is determined by number of bits used for page number: True

The page table is indexed using frame number: False

The page table is itself present in Physical memory: True

The page table is indexed using page number: True

The physical address may not be of the same size (in bits) as the logical address: True

The locating of the page table using PTBR also involves paging translation: False

Question 4

Correct

Mark 0.50 out of 0.50

Mark statements as True/False w.r.t. the creation of free page list in xv6.

True	False	
<input checked="" type="radio"/>	<input type="radio"/>	the kmem.lock is used by kfree() and kalloc() only.
<input type="radio"/>	<input checked="" type="radio"/>	free page list is a singly circular linked list.
<input checked="" type="radio"/>	<input type="radio"/>	if(kmem.use_lock) acquire(&kmem.lock); is not done when called from kinit1() because there is no need to take the lock when kinit1() is running because interrupts are disabled and only one processor is running
<input checked="" type="radio"/>	<input type="radio"/>	kmem.use_lock is set to 1 after free page list is created, so that kmem.lock is taken before accessing kmem.freelist.
<input type="radio"/>	<input checked="" type="radio"/>	if(kmem.use_lock) acquire(&kmem.lock); this "if" condition is true, when kinit2() runs because multi- processor support has been enabled by now.
<input checked="" type="radio"/>	<input type="radio"/>	The pointers that link the pages together are in the first 4 bytes of the pages themselves

the kmem.lock is used by kfree() and kalloc() only.: True

free page list is a singly circular linked list.: False

if(kmem.use_lock)

acquire(&kmem.lock);

is not done when called from kinit1() because there is no need to take the lock when kinit1() is running because interrupts are disabled and only one processor is running: True

kmem.use_lock is set to 1 after free page list is created, so that kmem.lock is taken before accessing kmem.freelist.: True

if(kmem.use_lock)

acquire(&kmem.lock);

this "if" condition is true, when kinit2() runs because multi-processor support has been enabled by now.: False

The pointers that link the pages together are in the first 4 bytes of the pages themselves: True

Question 5

Correct

Mark 0.50 out of 0.50

Map ext2 data structure features with their purpose

Mount count in superblock ✓

to enforce file check after certain amount of mounts at boot time

A group ✓
Combining file type and access rights in one variable ✓

Try to keep all the data of a directory and its file close together in a group

Block bitmap is one block ✓

Limits the size of a block group, thus improvising on purpose of a group

Inode bitmap is one block ✓

limits total number of files that can belong to a group

File Name is padded ✓

aligns all memory accesses on word boundary, improving performance

rec_len field in directory entry ✓

allows holes and linking of entries in directory

Inode table location in Group Descriptor ✓

Obvious, as it's per group and not per file-system

Inode table ✓
Used directories count in group descriptor ✓

All inodes are kept together so that one disk read leads to reading many inodes together, effectively doing a buffering of subsequent inode reads, and to save space on disk

Free blocks count in superblock and group descriptor ✓

Redundancy to help fsck restore consistency

Many copies of Superblock ✓

Redundancy to ensure the most crucial data structure is not lost

Your answer is correct.

The correct answer is: Mount count in superblock → to enforce file check after certain amount of mounts at boot time, A group → Try to keep all the data of a directory and its file close together in a group, Combining file type and access rights in one variable → saves 1 byte of space, Block bitmap is one block → Limits the size of a block group, thus improvising on purpose of a group, Inode bitmap is one block → limits total number of files that can belong to a group, File Name is padded → aligns all memory accesses on word boundary, improving performance, rec_len field in directory entry → allows holes and linking of entries in directory, Inode table location in Group Descriptor → Obvious, as it's per group and not per file-system, Inode table → All inodes are kept together so that one disk read leads to reading many inodes together, effectively doing a buffering of subsequent inode reads, and to save space on disk, Used directories count in group descriptor → attempt is made to evenly spread the first-level directories, this count is used there

attempt is made to evenly spread the first-level directories, this count is used there, Free blocks count in superblock and group descriptor → Redundancy to help fsck restore consistency, Many copies of Superblock → Redundancy to ensure the most crucial data structure is not lost

Question 6

Partially correct

Mark 0.43 out of 0.50

Given below are statements about concurrency and parallelism

Select T/F

A concurrent system can allow more than one task to progress, whereas a parallel system can perform more than one task at the same time.

True	False	
<input type="radio"/> ✗	<input checked="" type="radio"/> ✗	Parallel systems allow more than one task to progress while concurrent systems do not.
<input checked="" type="radio"/> ✗	<input type="radio"/> ✗	It is possible to have concurrency without parallelism
<input checked="" type="radio"/> ✗	<input type="radio"/> ✗	A concurrent system can allow more than one task to progress, whereas a parallel system can perform more than one task at the same time.
<input type="radio"/> ✗	<input checked="" type="radio"/> ✗	A concurrent system allows more than one task to progress while a parallel system does not.
<input type="radio"/> ✗	<input checked="" type="radio"/> ✗	Both concurrency and parallelism are the same.
<input type="radio"/> ✗	<input checked="" type="radio"/> ✗	It is possible to have parallelism without concurrency
<input type="radio"/> ✗	<input checked="" type="radio"/> ✗	It is not possible to have concurrency without parallelism.

Parallel systems allow more than one task to progress while concurrent systems do not.: False

It is possible to have concurrency without parallelism: True

A concurrent system can allow more than one task to progress, whereas a parallel system can perform more than one task at the same time.: True

A concurrent system allows more than one task to progress while a parallel system does not.: False

Both concurrency and parallelism are the same.: False

It is possible to have parallelism without concurrency: False

It is not possible to have concurrency without parallelism.: False

Question 7

Correct

Mark 0.50 out of 0.50

Which of the following is not a task of the code of swtch() function

- a. Switch stacks
- b. Save the return value of the old context code ✓
- c. Load the new context
- d. Jump to next context EIP
- e. Change the kernel stack location ✓
- f. Save the old context

The correct answers are: Save the return value of the old context code, Change the kernel stack location

Question 8

Correct

Mark 0.50 out of 0.50

Which of the following is DONE by allocproc() ?

- a. ensure that the process starts in trapret()
- b. ensure that the process starts in forkret()✓
- c. allocate PID to the process✓
- d. Select an UNUSED struct proc for use✓
- e. setup the contents of the trapframe of the process properly
- f. allocate kernel stack for the process✓
- g. setup the trapframe and context pointers appropriately✓
- h. setup kernel memory mappings for the process

The correct answers are: Select an UNUSED struct proc for use, allocate PID to the process, allocate kernel stack for the process, setup the trapframe and context pointers appropriately, ensure that the process starts in forkret()

Question 9

Correct

Mark 0.50 out of 0.50

Mark the statements as True or False, w.r.t. passing of arguments to system calls in xv6 code.

True	False	
<input checked="" type="radio"/>	<input type="radio"/>	Integer arguments are copied from user memory to kernel memory using argint()
<input checked="" type="radio"/>	<input type="radio"/>	The functions like argint(), argstr() make the system call arguments available in the kernel.
<input type="radio"/>	<input checked="" type="radio"/>	The arguments to system call are copied to kernel stack in trapasm.S
<input checked="" type="radio"/>	<input type="radio"/>	The arguments to system call originally reside on process stack.
<input type="radio"/>	<input checked="" type="radio"/>	Integer arguments are stored in eax, ebx, ecx, etc. registers
<input checked="" type="radio"/>	<input type="radio"/>	The arguments are accessed in the kernel code using esp on the trapframe.
<input checked="" type="radio"/>	<input type="radio"/>	String arguments are NOT copied in kernel memory, but just pointed to by a kernel memory pointer
<input type="radio"/>	<input checked="" type="radio"/>	String arguments are first copied to trapframe and then from trapframe to kernel's other variables.

Integer arguments are copied from user memory to kernel memory using argint(): True

The functions like argint(), argstr() make the system call arguments available in the kernel.: True

The arguments to system call are copied to kernel stack in trapasm.S: False

The arguments to system call originally reside on process stack.: True

Integer arguments are stored in eax, ebx, ecx, etc. registers: False

The arguments are accessed in the kernel code using esp on the trapframe.: True

String arguments are NOT copied in kernel memory, but just pointed to by a kernel memory pointer: True

String arguments are first copied to trapframe and then from trapframe to kernel's other variables.: False

Question 10

Incorrect

Mark 0.00 out of 0.50

In the code below assume that each function can be executed concurrently by many threads/processes.
Ignore syntactical issues, and focus on the semantics.

This program is an example of

```
spinlock a, b; // assume initialized
thread1() {
    spinlock(a);
    //some code;
    spinlock(b);
    //some code;
    spinunlock(b);
    spinunlock(a);
}
thread2() {
    spinlock(a);
    //some code;
    spinlock(b);
    //some code;
    spinunlock(b);
    spinunlock(a);
}
```

- a. Deadlock or livelock depending on actual race
- b. Self Deadlock
- c. Livelock
- d. DeadlockX
- e. None of these

Your answer is incorrect.

The correct answer is: None of these

Question 11

Correct

Mark 0.50 out of 0.50

Select the correct statements about paging (not demand paging) mechanism

Select one or more:

- a. Page table is accessed by the OS as part of execution of an instruction
- b. OS creates the page table for every process ✓
- c. User process can update its own PTBR
- d. Page table is accessed by the MMU as part of execution of an instruction ✓
- e. An invalid entry on a page means, it was an illegal memory reference ✓
- f. An invalid entry on a page means, either it was illegal memory reference or the page was not present in memory.
- g. The PTBR is loaded by the OS ✓
- h. User process can update its own page table entries

Your answer is correct.

The correct answers are: OS creates the page table for every process, The PTBR is loaded by the OS, Page table is accessed by the MMU as part of execution of an instruction, An invalid entry on a page means, it was an illegal memory reference

Question 12

Correct

Mark 0.50 out of 0.50

Why V2P_WO is used in entry.S and not V2P ?

- a. It's a mistake. They could have used the same macro in both places.
- b. The two macros are different. They lead to different calculations.
- c. Because the processor can not do a type casting at run time
- d. Because entry.S is an assembly code file and assemblers do not know about data types and type casting. ✓
- e. The typecasting has the effect of creating virtual address, while without typecast we get physical address.

Your answer is correct.

The correct answer is: Because entry.S is an assembly code file and assemblers do not know about data types and type casting.

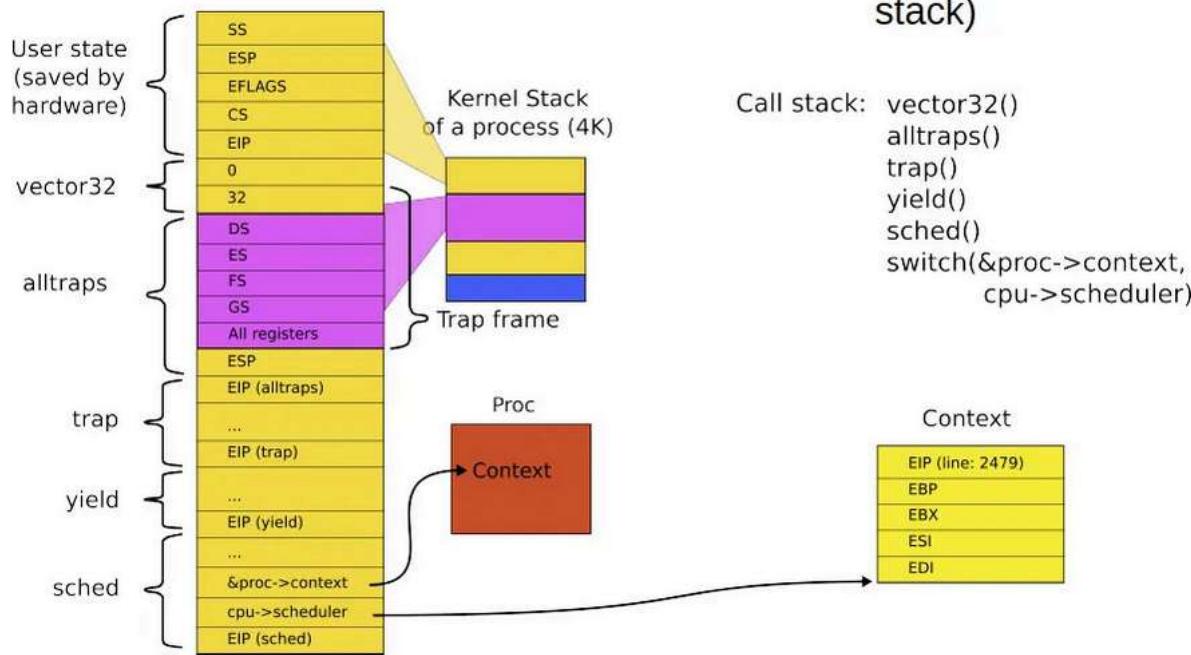
Question 13

Correct

Mark 0.50 out of 0.50

Mark statements as True/False, w.r.t. the given diagram

Stack inside swtch() and its two arguments (passed on the stack)


True False

<input checked="" type="radio"/> <input type="radio"/>	The diagram is wrong because it shows the user stack and kernel stack together (continuous), but in practice they are separate	<input checked="" type="checkbox"/>	diagram shows only kernel stack
<input checked="" type="checkbox"/> <input checked="" type="radio"/>	The diagram is correct	<input checked="" type="checkbox"/>	
<input checked="" type="radio"/> <input checked="" type="checkbox"/>	The "context" yellow coloured box, pointed to by cpu->scheduler is on the kernel stack of the scheduler.	<input checked="" type="checkbox"/>	
<input checked="" type="checkbox"/> <input checked="" type="radio"/>	The blue shaded part in "kernel stack of a process(4k)" refers to remaining part of stack (not used yet)	<input checked="" type="checkbox"/>	
<input checked="" type="radio"/> <input checked="" type="checkbox"/>	This is a diagram of swtch() called from scheduler()	<input checked="" type="checkbox"/>	No. diagram of swtch() called from sched()
<input checked="" type="checkbox"/> <input checked="" type="radio"/>	The "ESP" (second entry from top) is stack pointer of user-stack of process, while the "ESP" (first entry below pink region) is the trapframe pointer on kernel stack of process.	<input checked="" type="checkbox"/>	

The diagram is wrong because it shows the user stack and kernel stack together (continuous), but in practice they are separate: False

The diagram is correct: True

The "context" yellow coloured box, pointed to by cpu->scheduler is on the kernel stack of the scheduler.: False

The blue shaded part in "kernel stack of a process(4k)" refers to remaining part of stack (not used yet): True

This is a diagram of swtch() called from scheduler(): False

The "ESP" (second entry from top) is stack pointer of user-stack of process, while the "ESP" (first entry below pink region) is the trapframe pointer on kernel stack of process.: True

Question 14

Correct

Mark 0.50 out of 0.50

Mark the statements as True/False, with respect to the use of the variable "chan" in struct proc.

True	False	
<input type="radio"/> ✗	<input checked="" type="radio"/> ✗	chan is the head pointer to a linked list of processes, waiting for a particular event to occur
<input checked="" type="radio"/> ✗	<input type="radio"/> ✗	When chan is not NULL, the 'state' in struct proc must be SLEEPING
<input checked="" type="radio"/> ✗	<input type="radio"/> ✗	chan stores the address of the variable, representing a condition, for which the process is waiting.
<input checked="" type="radio"/> ✗	<input type="radio"/> ✗	The value of 'chan' is changed only in sleep()
<input checked="" type="radio"/> ✗	<input type="radio"/> ✗	in xv6, the address of an appropriate variable is used as a "condition" for a waiting process.
<input type="radio"/> ✗	<input checked="" type="radio"/> ✗	Changing the state of a process automatically changes the value of 'chan'
<input checked="" type="radio"/> ✗	<input type="radio"/> ✗	'chan' is used only by the sleep() and wakeup1() functions.
<input type="radio"/> ✗	<input checked="" type="radio"/> ✗	when chan is NULL, the 'state' in proc must be RUNNABLE.

chan is the head pointer to a linked list of processes, waiting for a particular event to occur.: False

When chan is not NULL, the 'state' in struct proc must be SLEEPING.: True

chan stores the address of the variable, representing a condition, for which the process is waiting.: True

The value of 'chan' is changed only in sleep(): True

in xv6, the address of an appropriate variable is used as a "condition" for a waiting process.: True

Changing the state of a process automatically changes the value of 'chan': False

'chan' is used only by the sleep() and wakeup1() functions.: True

when chan is NULL, the 'state' in proc must be RUNNABLE.: False

Question 15

Correct

Mark 0.50 out of 0.50

Doing a lookup on the pathname /a/b/b/c/d for opening the file "d" requires reading no. of inodes. Assume that there are no hard/soft links on the path.

Write the answer as a number.

The correct answer is: 6

Question 16

Partially correct

Mark 0.25 out of 0.50

Suppose a file is to be created in an ext2 file system, in an existing directory /a/b/. Select from below, the list of blocks that may need modification.

Select one or more:

- a. inode of /a/b/✓
- b. inode of /a/
- c. link count on /a/b/ inode
- d. inode bitmap in some block group
- e. group descriptor(s)✓
- f. inode bitmap referring to /a/b/
- g. new data block in some block group
- h. superblock✓
- i. data blocks of /a/
- j. inode table in some block group
- k. block bitmap in some block group✓
- l. existing data blocks of /a/b/

Your answer is partially correct.

You have correctly selected 4.

The correct answers are: superblock, group descriptor(s), inode of /a/b/, existing data blocks of /a/b/, inode table in some block group, inode bitmap in some block group, block bitmap in some block group, new data block in some block group

Question 17

Partially correct

Mark 0.33 out of 0.50

Select the correct statements about hard and soft links

Select one or more:

- a. Soft link shares the inode of actual file
- b. Deleting a soft link deletes only the actual file
- c. Deleting a soft link deletes the link, not the actual file
- d. Deleting a soft link deletes both the link and the actual file ✗
- e. Hard links share the inode ✓
- f. Hard links enforce separation of filename from its metadata in on-disk data structures. ✓
- g. Soft links can span across partitions while hard links can't ✓
- h. Hard links can span across partitions while soft links can't
- i. Deleting a hard link deletes the file, only if link count was 1 ✓
- j. Soft links increase the link count of the actual file inode
- k. Hard links increase the link count of the actual file inode ✓
- l. Deleting a hard link always deletes the file

Your answer is partially correct.

You have correctly selected 5.

The correct answers are: Soft links can span across partitions while hard links can't, Hard links increase the link count of the actual file inode, Deleting a soft link deletes the link, not the actual file, Deleting a hard link deletes the file, only if link count was 1, Hard links share the inode, Hard links enforce separation of filename from its metadata in on-disk data structures.

Question 18

Correct

Mark 0.50 out of 0.50

It is proposed that when a process does an illegal memory access, xv6 terminate the process by printing the error message "Illegal Memory Access". Select all the changes that need to be done to xv6 for this as True (Note that the changes proposed here may not cover the exhaustive list of all changes required) and the unnecessary/wrong changes as False.

Required	Un-necessary/Wrong	
<input checked="" type="radio"/>	<input type="radio"/> <input checked="" type="checkbox"/>	Change in the Makefile and instruct cc/ld to start the code of each program at some address other than 0
<input checked="" type="radio"/>	<input type="radio"/> <input checked="" type="checkbox"/>	Change allocuvm() to call mappages() with proper permissions on each page table entry
<input type="radio"/> <input checked="" type="checkbox"/>	<input type="radio"/>	Change mappages() to set specified permissions on each page table entry
<input type="radio"/> <input checked="" type="checkbox"/>	<input checked="" type="radio"/>	Mark each page as readonly in the page table mappings
<input checked="" type="radio"/>	<input type="radio"/> <input checked="" type="checkbox"/>	Ensure that the address 0 is mapped to invalid
<input checked="" type="radio"/>	<input type="radio"/>	Handle the Illegal memory access trap in trap() function, and terminate the currently running process.
<input checked="" type="radio"/>	<input type="radio"/> <input checked="" type="checkbox"/>	Change exec to treat text/data sections separately and call allocuvm() with proper flags for page table entries
<input type="radio"/> <input checked="" type="checkbox"/>	<input checked="" type="radio"/>	Add code that checks if the illegal memory access trap was due to an actual illegal memory access.

Change in the Makefile and instruct cc/ld to start the code of each program at some address other than 0: Required

Change allocuvm() to call mappages() with proper permissions on each page table entry: Required

Change mappages() to set specified permissions on each page table entry: Un-necessary/Wrong

Mark each page as readonly in the page table mappings: Un-necessary/Wrong

Ensure that the address 0 is mapped to invalid: Required

Handle the Illegal memory access trap in trap() function, and terminate the currently running process.: Required

Change exec to treat text/data sections separately and call allocuvm() with proper flags for page table entries: Required

Add code that checks if the illegal memory access trap was due to an actual illegal memory access.: Un-necessary/Wrong

Question 19

Correct

Mark 0.50 out of 0.50

The variable 'end' used as argument to kinit1 has the value

- a. 8010a48c
- b. 80110000
- c. 80000000
- d. 801154a8 ✓
- e. 80102da0
- f. 81000000

The correct answer is: 801154a8

Question 20

Partially correct

Mark 0.33 out of 0.50

Mark statements about deadlocks as True or false

True	False	
<input type="radio"/> ✗	<input checked="" type="radio"/> ✗	Deadlocks are the same as livelocks
<input checked="" type="radio"/> ✗	<input type="radio"/> ✗	A deadlock is possible only if all the 4 conditions of mutual exclusion, cyclic wait, hold and wait, and no preemption are satisfied
<input checked="" type="radio"/> ✗	<input type="radio"/> ✗	A deadlock necessarily requires a cycle in the resource allocation graph
<input checked="" type="radio"/> ✗	<input type="radio"/> ✗	Cycle in the resource allocation graph does not necessarily mean a deadlock
<input type="radio"/> ✗	<input checked="" type="radio"/> ✗	A deadlock must involve at least two processes
<input checked="" type="radio"/> ✗	<input type="radio"/> ✗	Deadlocks are not possible if there is no race

Deadlocks are the same as livelocks: False

A deadlock is possible only if all the 4 conditions of mutual exclusion, cyclic wait, hold and wait, and no preemption are satisfied: True

A deadlock necessarily requires a cycle in the resource allocation graph: True

Cycle in the resource allocation graph does not necessarily mean a deadlock: True

A deadlock must involve at least two processes: False

Deadlocks are not possible if there is no race: True

Question 21

Correct

Mark 0.50 out of 0.50

Select the statement that most correctly describes what setupkvm() does

- a. creates a 1-level page table for the use by the kernel, as specified in kmap[] global array
- b. creates a 2-level page table setup with virtual->physical mappings specified in the kmap[] global array ✓
- c. creates a 2-level page table setup with virtual->physical mappings specified in the kmap[] global array and makes kpgdir point to it
- d. creates a 2-level page table for the use of the kernel, as specified in gdtdesc

The correct answer is: creates a 2-level page table setup with virtual->physical mappings specified in the kmap[] global array

Question 22

Correct

Mark 0.50 out of 0.50

Mark statements as T/F

All statements are in the context of preventing deadlocks.

True	False	
<input checked="" type="radio"/>	<input type="radio"/> ✗	A process holding one resources and waiting for just one more resource can also be involved in a deadlock.
<input checked="" type="radio"/>	<input type="radio"/> ✗	Mutual exclusion is a necessary condition for deadlock because it brings in locks on which deadlock happens
<input type="radio"/> ✗	<input checked="" type="radio"/>	The lock ordering to be followed to avoid circular wait is a code in OS that checks for compliance with decided order
<input checked="" type="radio"/>	<input type="radio"/> ✗	Hold and wait means a thread/process holding some locks and waiting for acquiring some.
<input type="radio"/> ✗	<input checked="" type="radio"/>	If a resource allocation graph contains a cycle then there is a guarantee of a deadlock
<input checked="" type="radio"/>	<input type="radio"/> ✗	Circular wait is avoided by enforcing a lock ordering
<input checked="" type="radio"/>	<input type="radio"/> ✗	Deadlock is possible if all the conditions are met at the same time: Mutual exclusion, hold and wait, no pre-emption, circular wait.

A process holding one resources and waiting for just one more resource can also be involved in a deadlock.: True

Mutual exclusion is a necessary condition for deadlock because it brings in locks on which deadlock happens: True

The lock ordering to be followed to avoid circular wait is a code in OS that checks for compliance with decided order: False

Hold and wait means a thread/process holding some locks and waiting for acquiring some: True

If a resource allocation graph contains a cycle then there is a guarantee of a deadlock: False

Circular wait is avoided by enforcing a lock ordering: True

Deadlock is possible if all the conditions are met at the same time: Mutual exclusion, hold and wait, no pre-emption, circular wait.: True

Question 23

Correct

Mark 0.50 out of 0.50

Will this code work for a spinlock() operation? The intention here is to call compare-and-swap() only if the lock is not held (the if condition checks for the same).

```
void spinlock(int *lock) {
{
    while (true) {
        if (*lock == 0) {
            /* lock appears to be available */
            if (!compare_and_swap(lock, 0, 1))
                break
        }
    }
}
```

- a. No, because in the case of both processes succeeding in the "if" condition, both may end up acquiring the lock.
- b. Yes, because no matter in which order the if-check and compare-and-swap run in multiple processes, only one process will succeed in compare-and-swap() and others will keep looping in while-loop. ✓
- c. Yes, because there is no race to update the lock variable
- d. No, because this breaks the atomicity requirement of compare-and-test.

Your answer is correct.

The correct answer is: Yes, because no matter in which order the if-check and compare-and-swap run in multiple processes, only one process will succeed in compare-and-swap() and others will keep looping in while-loop.

Question 24

Partially correct

Mark 0.38 out of 0.50

The kernel ELF file contains these headers

Program Header:

```
LOAD off 0x00001000 vaddr 0x80100000 paddr 0x00100000 align 2**12
  filesz 0x00007aab memsz 0x00007aab flags r-x
LOAD off 0x00009000 vaddr 0x80108000 paddr 0x00108000 align 2**12
  filesz 0x00002516 memsz 0x0000d4a8 flags rw-
STACK off 0x00000000 vaddr 0x00000000 paddr 0x00000000 align 2**4
  filesz 0x00000000 memsz 0x00000000 flags rwx
```

mark the statements as True/False

True	False	
<input checked="" type="radio"/>	<input type="radio"/> <input checked="" type="checkbox"/>	Second header is for Data/Globals
<input checked="" type="radio"/>	<input type="radio"/> <input checked="" type="checkbox"/>	First header is for the code/text
<input type="radio"/>	<input checked="" type="radio"/> <input checked="" type="checkbox"/>	in bootmain() the third header leads to allocation of no-memory.
<input checked="" type="radio"/>	<input type="radio"/> <input checked="" type="checkbox"/>	Third header is for stack

Second header is for Data/Globals: True

First header is for the code/text: True

in bootmain() the third header leads to allocation of no-memory.: True

Third header is for stack: True

Question 25

Correct

Mark 0.50 out of 0.50

Consider this program.

Some statements are identified using the // comment at the end.

Assume that = is an atomic operation.

```
#include <stdio.h>
#include <pthread.h>
long c = 0, c1 = 0, c2 = 0, run = 1;
void *thread1(void *arg) {
    while(run == 1) { //E
        c = 10; //A
        c1 = c2 + 5; //B
    }
}
void *thread2(void *arg) {
    while(run == 1) { //F
        c = 20; //C
        c2 = c1 + 3; //D
    }
}
int main() {
    pthread_t th1, th2;
    pthread_create(&th1, NULL, thread1, NULL);
    pthread_create(&th2, NULL, thread2, NULL);
    sleep(2);
    run = 0;
    fprintf(stdout, "c = %ld c1+c2 = %ld c1 = %ld c2 = %ld \n", c, c1+c2, c1, c2);
    fflush(stdout);
}
```

Which statements are part of the critical Section?

Yes	No	
<input type="radio"/> ✗	<input checked="" type="radio"/>	C
<input checked="" type="radio"/>	<input type="radio"/> ✗	D
<input type="radio"/> ✗	<input checked="" type="radio"/>	A
<input type="radio"/> ✗	<input checked="" type="radio"/>	F
<input checked="" type="radio"/>	<input type="radio"/> ✗	B
<input type="radio"/> ✗	<input checked="" type="radio"/>	E

C: No

D: Yes

A: No

F: No

B: Yes

E: No

Question 26

Partially correct

Mark 0.38 out of 0.50

Match the code with its functionality

S = 0

P1:

Statement1;

Signal(S)

Execution order P1, then P2 ✓

P2:

Wait(S)

Statement2;

S1 = 0; S2 = 0;

P2:

Statement1;

Signal(S2);

P1:

Wait(S2);

Statement2;

Signal(S1);

Execution order P3, P2, P1 ✗

P3:

Wait(S1);

Statement S3;

S = 1

Wait(S)

Critical Section

Binary Semaphore for mutual exclusion ✓

Signal(S);

S = 5

Wait(S)

Critical Section

Counting semaphore ✓

Signal(S)

Your answer is partially correct.

You have correctly selected 3.

The correct answer is: S = 0

P1:

Statement1;

Signal(S)

✓

P2:

Wait(S)

Statement2; → Execution order P1, then P2, S1 = 0; S2 = 0;

P2:

Statement1;

Signal(S2);

✓

P1:

Wait(S2);

Statement2;

Signal(S1);

✓

P3:

Wait(S1);

Statement S3; → Execution order P2, P1, P3, S = 1

Wait(S)

Critical Section

Signal(S); → Binary Semaphore for mutual exclusion, S = 5

Wait(S)

✓

Critical Section

Signal(S) → Counting semaphore

Question 27

Correct

Mark 0.50 out of 0.50

In an ext2 file system, if the block size is 4KB and partition size is 128 GB, then the number of block groups will be:

Answer: 1024 ✓

size * 1024 * 1024 / 4 --> no of blocks

each group = 8 * 4 * 1024 blocks = 32768 blocks

so size * 1024 * 1024 / (4 * 32768) number of groups

The correct answer is: 1024.00

Question 28

Correct

Mark 0.50 out of 0.50

The "push 0" in vectors.S is

- a. Place for the error number value ✓
- b. To be filled in as the return value of the system call
- c. A placeholder to match the size of struct trapframe
- d. To indicate that it's a system call and not a hardware interrupt

The correct answer is: Place for the error number value

Question 29

Partially correct

Mark 0.25 out of 0.50

Match pairs

mutex	atomic test and set with loop	✗
peterson	per process flag, global turn variable	✓
semaphore	wait() and signal()	✓
spinlock	lock() and unlock()	✗

Your answer is partially correct.

You have correctly selected 2.

The correct answer is: mutex → lock() and unlock(), peterson → per process flag, global turn variable, semaphore → wait() and signal(), spinlock → atomic test and set with loop

Question **30**

Correct

Mark 0.50 out of 0.50

Which of the following is done by mappages()?

- a. allocate page frame if required
- b. allocate page directory if required
- c. create page table mappings to the range given by "pa" and "pa + size" ✓
- d. allocate page table if required ✓
- e. create page table mappings for the range given by "va" and "va + size" ✓

The correct answers are: create page table mappings for the range given by "va" and "va + size", allocate page table if required, create page table mappings to the range given by "pa" and "pa + size"

[◀ Quiz-1 \(15 Marks\)](#)

Jump to...

[ESE\(60 Marks\) ►](#)

Processes in xv6 code

Process Table

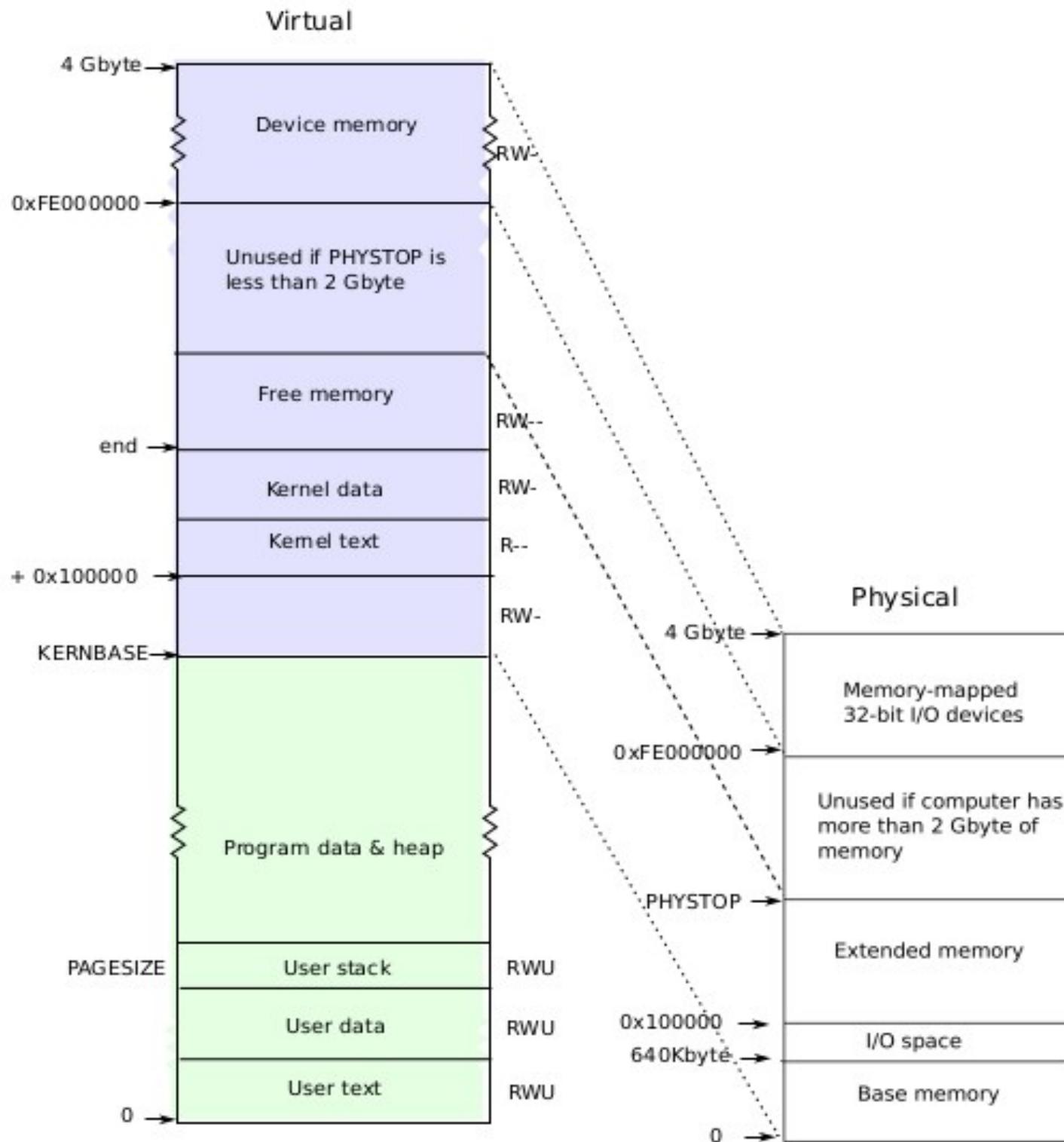
```
struct {  
    struct spinlock lock;  
    struct proc proc[NPROC];  
} ptable;
```

- One single global array of processes
- Protected by `ptable.lock`

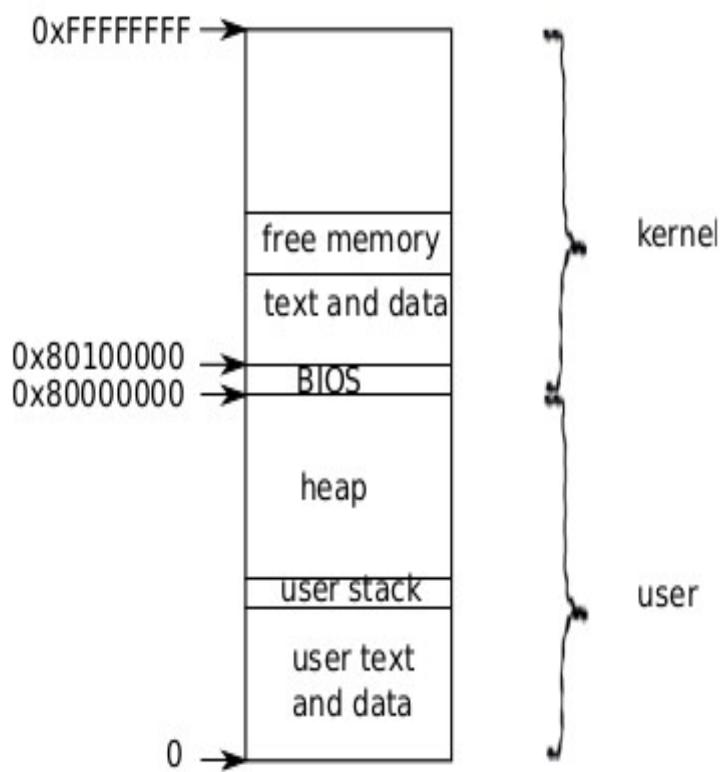
Layout of process's VA space

xv6
schema!

different
from Linux

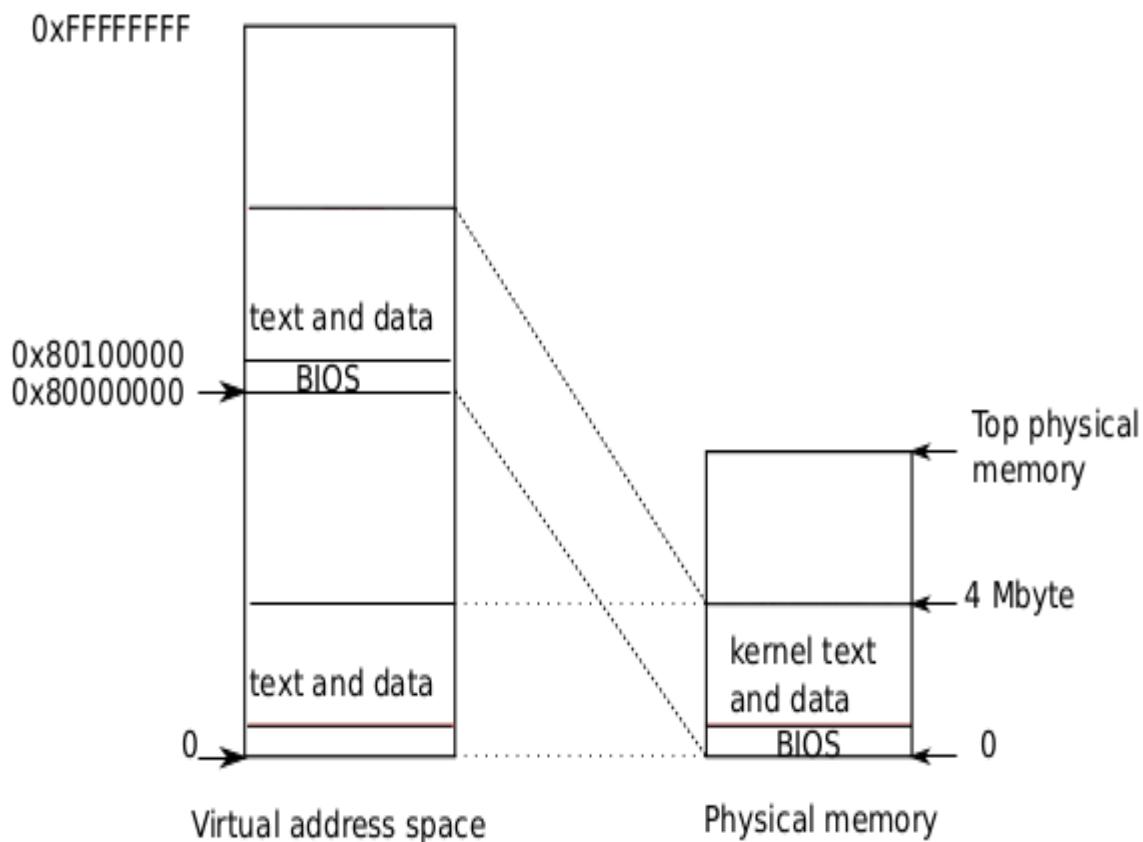


Logical layout of memory for a process



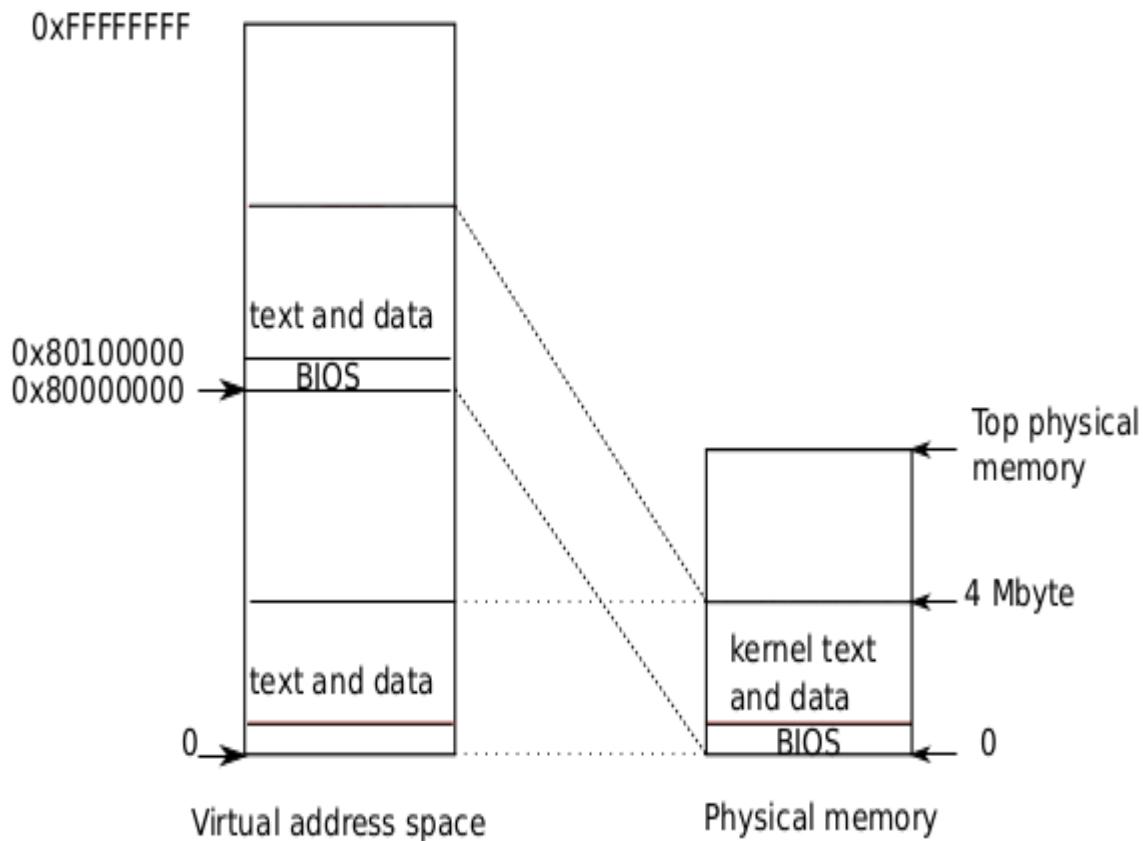
- **Address 0: code**
- **Then globals**
- **Then stack**
- **Then heap**
- **Each process's address space maps kernel's text, data also --> so that system calls run with these mappings**
- **Kernel code can directly access user memory now**

Kernel mappings in user address space actual location of kernel



- Kernel is loaded at **0x100000 physical address**
- PA 0 to 0x100000 is **BIOS and devices**
- Process's page table will map **VA 0x80000000 to PA 0x00000 and VA 0x80100000 to 0x100000**

Kernel mappings in user address space actual location of kernel



- Kernel is not loaded at the PA 0x80000000 because some systems may not have that much memory
- 0x80000000 is called **KERNBASE** in xv6

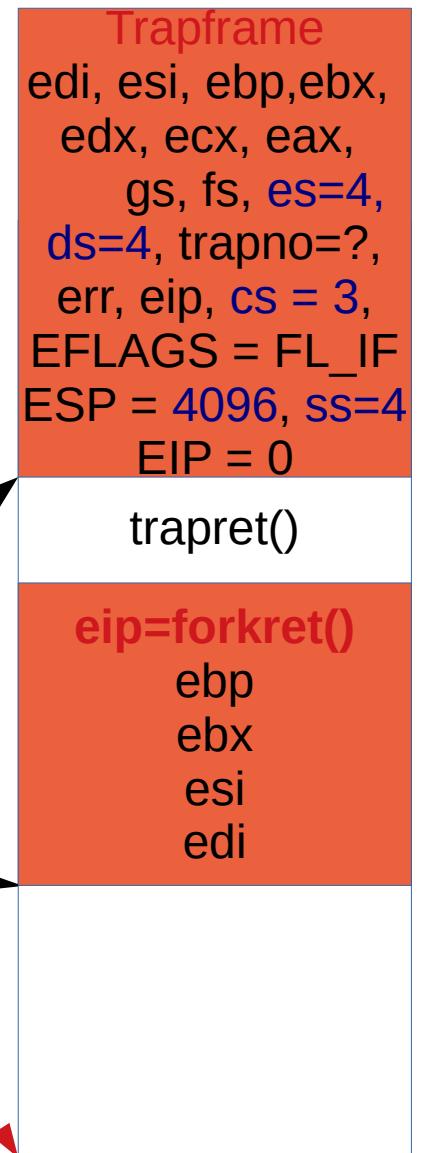
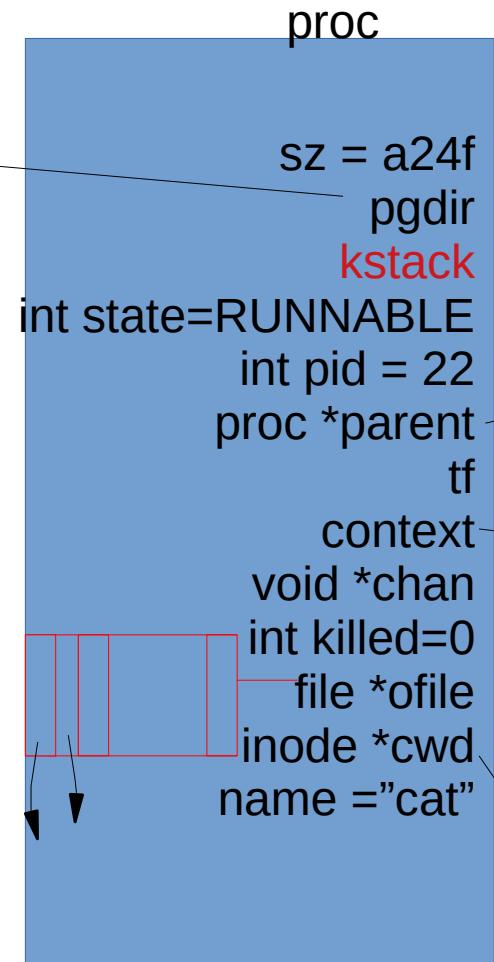
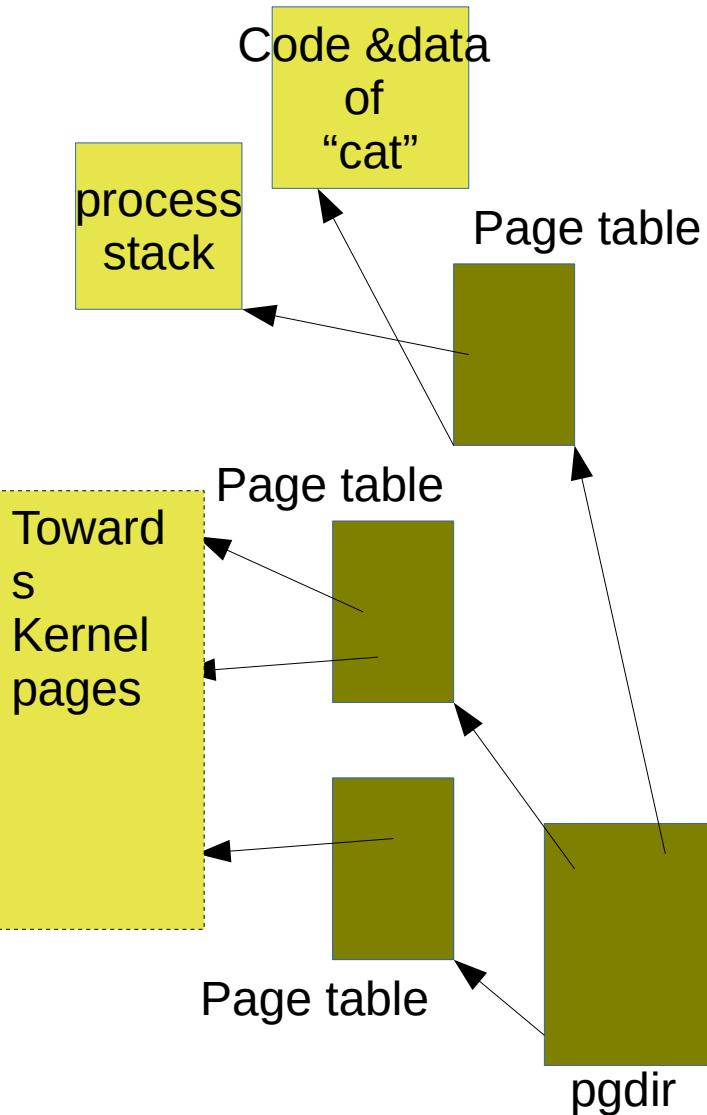
Imp Concepts

- **A process has two stacks**
 - user stack: used when user code is running
 - kernel stack: used when kernel is running on behalf of a process
- **Note: there is a third stack also!**
 - The kernel stack used by the scheduler itself
 - Not a per process stack

Struct proc

```
// Per-process state
struct proc {
    uint sz;                      // Size of process memory (bytes)
    pde_t* pgdir;                 // Page table
    char *kstack;                 // Bottom of kernel stack for this process
    enum procstate state;         // Process state. allocated, ready to run, running, waiting for I/O, or exiting.
    int pid;                      // Process ID
    struct proc *parent;          // Parent process
    struct trapframe *tf;         // Trap frame for current syscall
    struct context *context;      // swtch() here to run process. Process's context
    void *chan;                   // If non-zero, sleeping on chan. More when we discuss sleep, wakeup
    int killed;                   // If non-zero, have been killed
    struct file *ofile[NOFILE];   // Open files, used by open(), read(),...
    struct inode *cwd;            // Current directory, changed with "chdir()"
    char name[16];                // Process name (for debugging)
};
```

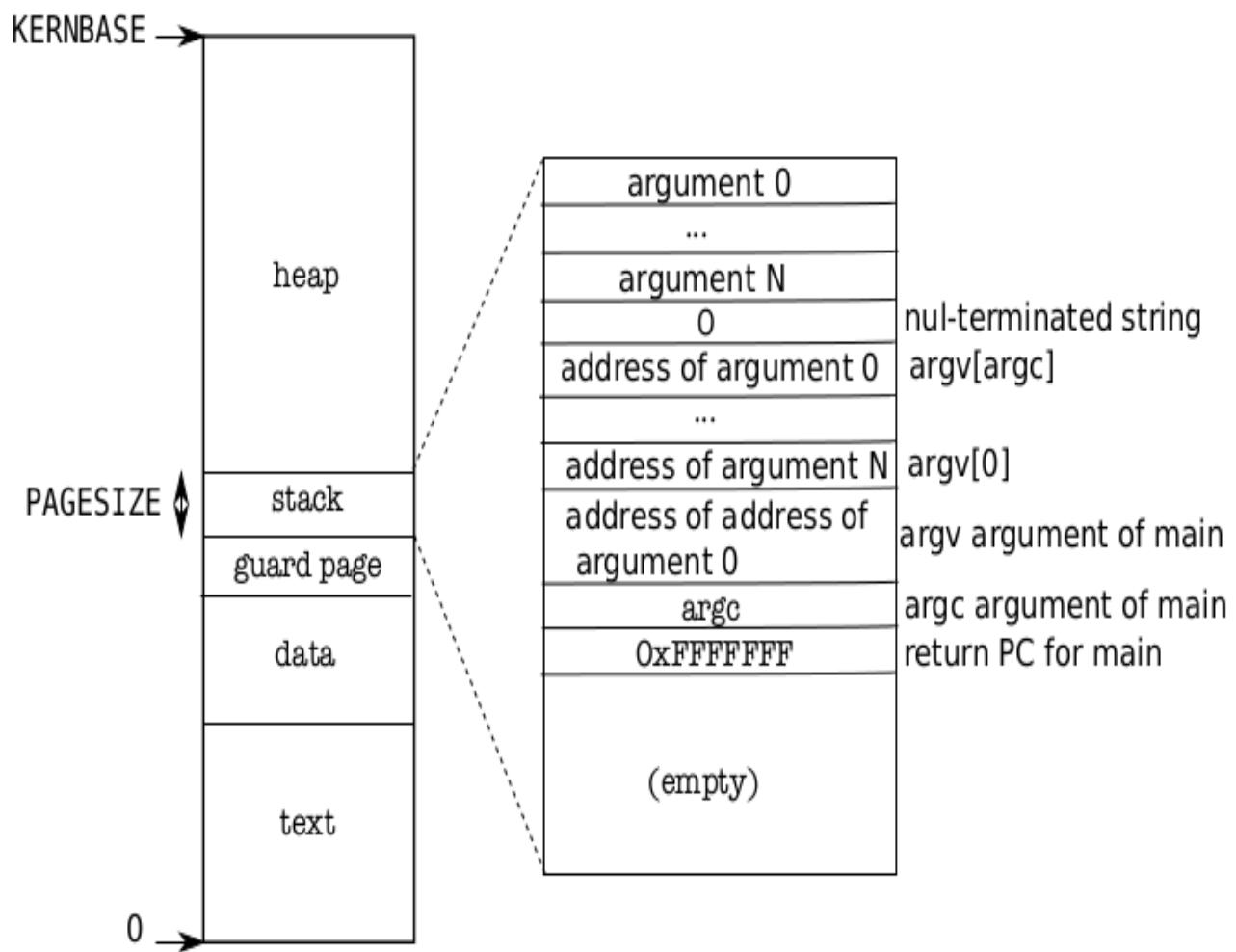
struct proc diagram: Very imp!



$sz = \text{ELF-code-} \rightarrow \text{memsz}$ (includes data, check "ld -N"
+ 2×4096 (for stack)

In use only when you are in kernel on a "trap" = interrupt/syscall. "tf" always used. trapret,forkret used during fork()

Memory Layout of a user process



Memory Layout of a user process

After exec()

Note the argc, argv on stack

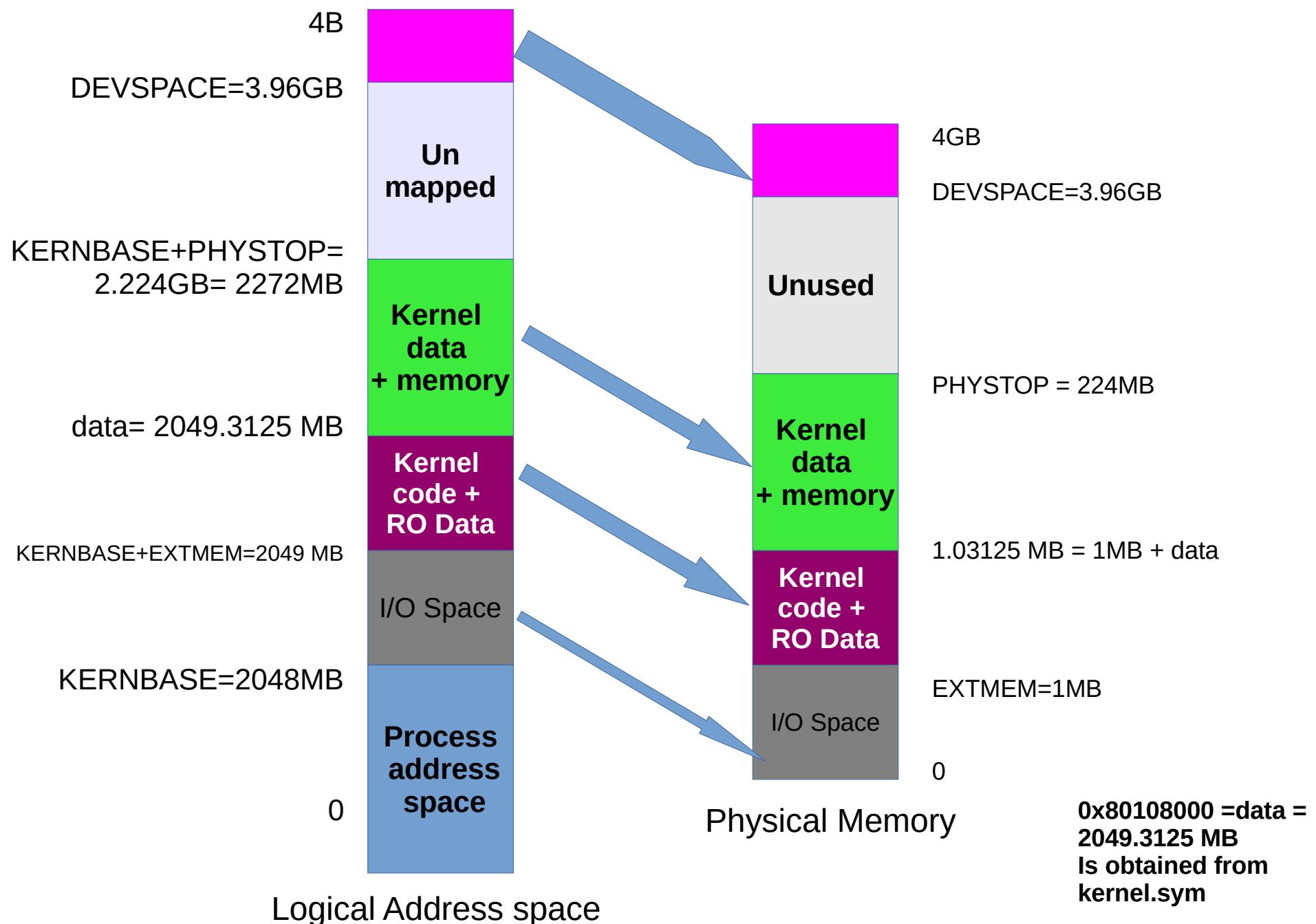
The “guard page” is just a mapping in page table. No frame allocated. It’s marked as invalid. So if stack grows (due to many function calls), then OS will detect it with an exception

Handling Traps

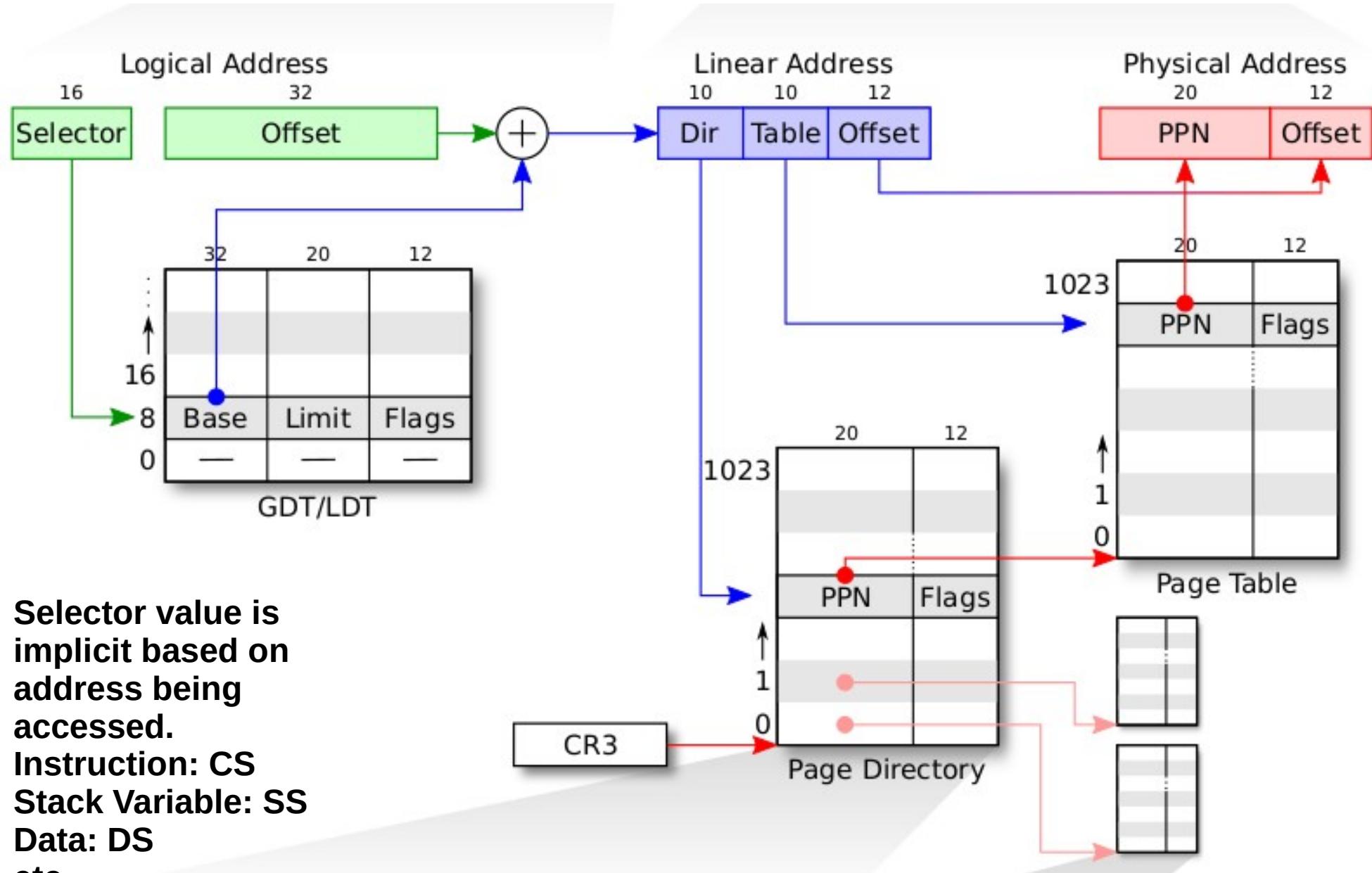
Some basic steps

- **Xv6.img is created by “make”**
 - Contains bootsector, kernel code, kernel data
- **QEMU boots using xv6.img**
 - First it runs bootloader
 - Bootloader loads kernel of xv6 (from xv6.img)
 - Kernel starts running
- **Kernel running..**
 - Kernel calls main() of kernel (NOT a C application!) & Initializes:
 - memory management data structures
 - process data structures
 - file handling data structures
 - Multi-processors
 - Multi-processor data structures
 - Interrupt Descriptor Table
 - ...
 - Then creates init()
 - Init() fork-execs shell

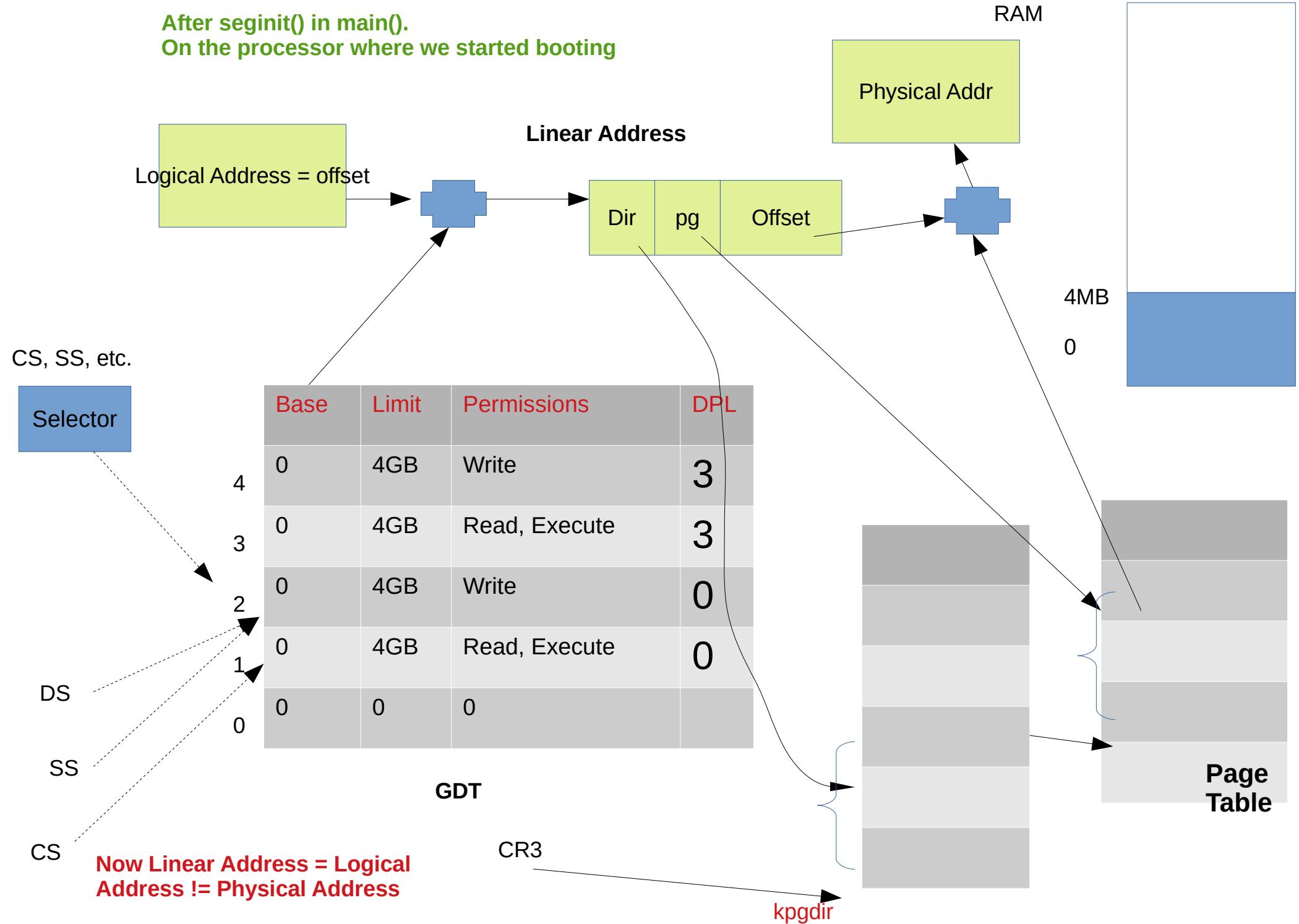
kmap[] mappings done in kvmalloc(). This shows segmentwise, entries are done in page directory and page table for corresponding VA->PA mappings



Segmentation + Paging



After seginit() in main().
On the processor where we started booting



Handling traps

- **Transition from user mode to kernel mode**
 - On a system call
 - On a hardware interrupt
 - User program doing illegal work (exception)
- **Actions needed, particularly w.r.t. to hardware interrupts**
 - Change to kernel mode & switch to kernel stack
 - Kernel to work with devices, if needed
 - Kernel to understand interface of device

Handling traps

- **Actions needed on a trap**
 - Save the processor's registers (**context**) for future use
 - Set up the system to run kernel code (**kernel context**) on kernel stack
 - Start kernel in appropriate place (**sys call, intr handler, etc**)
 - Kernel to get all info related to event (**which block I/O done?, which sys call called, which process did exception and what type, get arguments to system call, etc**)

Privilege level

- The x86 has 4 protection levels, numbered 0 (most privilege) to 3 (least privilege).
- In practice, most operating systems use only 2 levels: 0 and 3, which are then called kernel mode and user mode, respectively.
- The current privilege level with which the x86 executes instructions is stored in %cs register, in the field CPL.



TI Table index (0=GDT, 1=LDT)
RPL Requester privilege level

Privilege level

- **Changes automatically on**
 - “int” instruction
 - hardware interrupt
 - exception
- **Changes back on**
 - iret
- **“int” 10 --> makes 10th hardware interrupt. S/w interrupt can be used to ‘create hardware interrupt’**
- **Xv6 uses “int 64” for actual system calls**

Interrupt Descriptor Table (IDT)

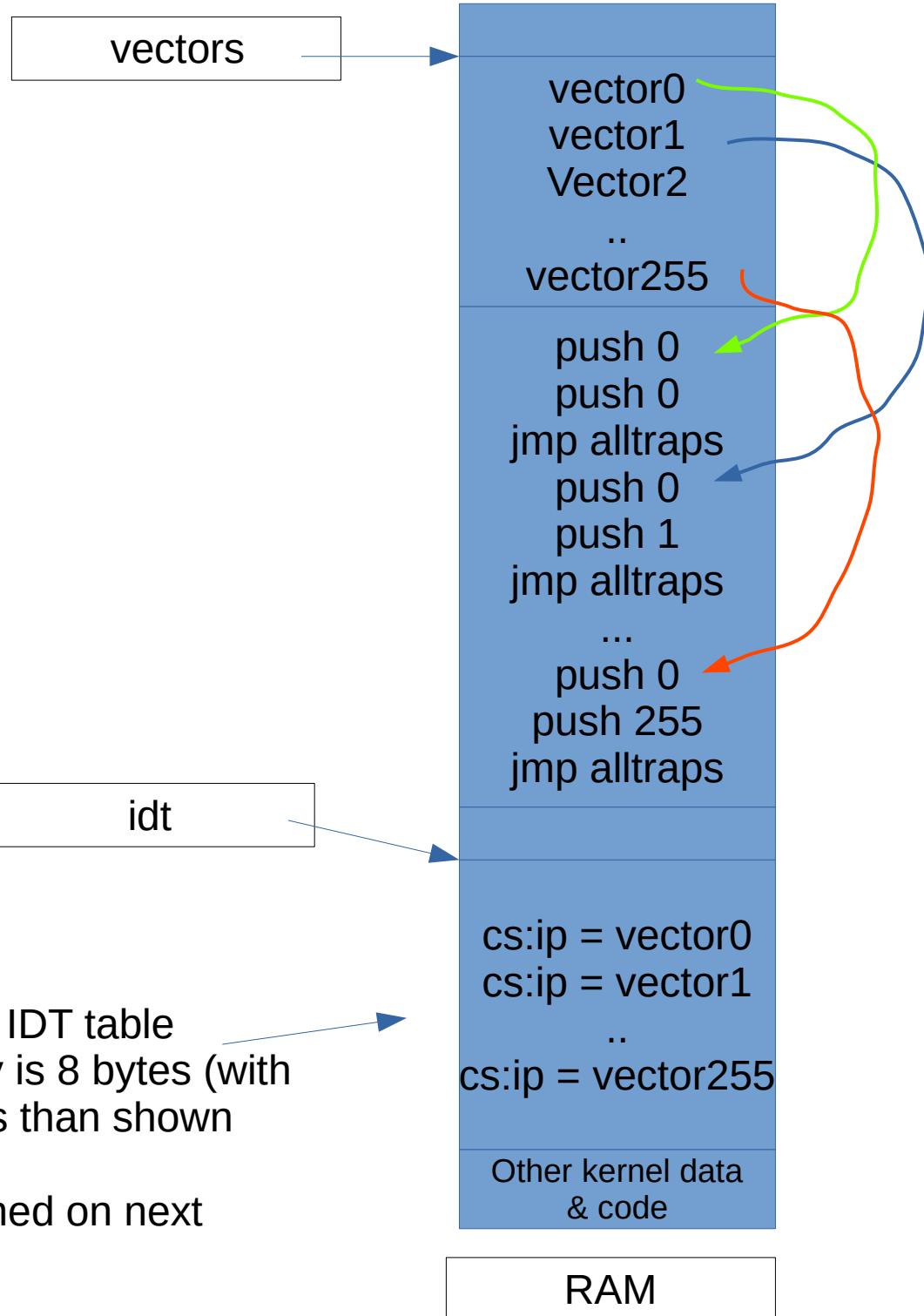
- **IDT defines interrupt handlers**
- **Has 256 entries**
 - each giving the %cs and %eip to be used when handling the corresponding interrupt.
- **Mapping**
 - Interrupts 0-31 are defined for software exceptions, like divide errors or attempts to access invalid memory addresses.
 - Xv6 maps the 32 hardware interrupts to the range 32-63
 - and uses interrupt 64 as the system call interrupt

IDT setup done by tvinit() function

The array of “vectors”
And the code of
“push, ..jmp”
Is part of kernel image
(xv6.img)

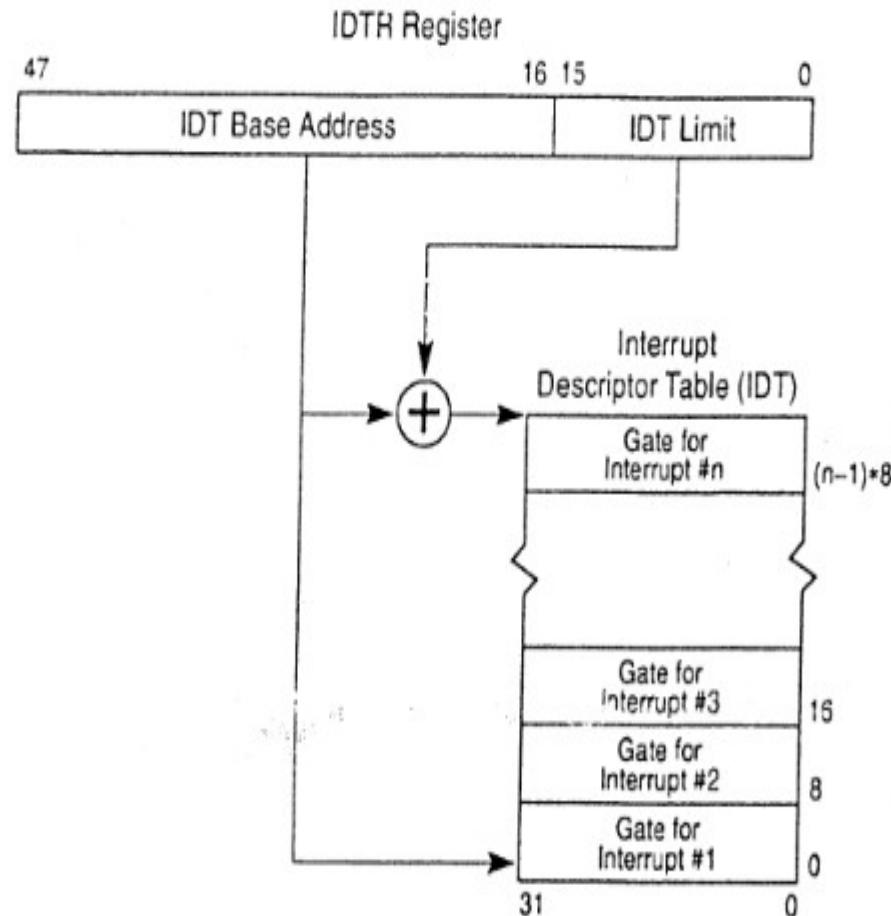
The tvinit() is called during
kernel initialization

This is the IDT table
Each entry is 8 bytes (with
more fields than shown
here)
as mentioned on next
slides



RAM

IDTR and IDT



IDT is in RAM

IDTR is in CPU

Interrupt Descriptor Table (IDT) entries (in RAM)

```
// Gate descriptors for interrupts and traps

struct gatedesc {

    uint off_15_0 : 16; // low 16 bits of offset in segment

    uint cs : 16; // code segment selector

    uint args : 5; // # args, 0 for interrupt/trap gates

    uint rsv1 : 3; // reserved(should be zero I guess)

    uint type : 4; // type(STS_{IG32,TG32})

    uint s : 1; // must be 0 (system)

    uint dpl : 2; //descriptor(new) privilege level

    uint p : 1; // Present

    uint off_31_16 : 16; // high bits of offset in segment

};
```

Setting IDT entries

```
void
tvinit(void)
{
    int i;
    for(i = 0; i < 256; i++)
        SETGATE(idt[i], 0, SEG_KCODE<<3, vectors[i], 0);
    SETGATE(idt[T_SYSCALL], 1, SEG_KCODE<<3,
             vectors[T_SYSCALL], DPL_USER);
    /* value 1 in second argument --> don't disable
    interrupts
       * DPL_USER means that processes can raise
    this interrupt. */
    initlock(&tickslock, "time");
}
```

Setting IDT entries

```
#define SETGATE(gate, istrap, sel, off, d) \
{ \
    (gate).off_15_0 = (uint)(off) & 0xffff; \
    (gate).cs = (sel); \
    (gate).args = 0; \
    (gate).rsv1 = 0; \
    (gate).type = (istrap) ? STS_TG32 : STS_IG32; \
    (gate).s = 0; \
    (gate).dpl = (d); \
    (gate).p = 1; \
    (gate).off_31_16 = (uint)(off) >> 16; \
}
```

Setting IDT entries

vectors.S

```
# generated by vectors.pl -  
do not edit  
  
# handlers  
  
.globl alltraps  
  
.globl vector0  
  
vector0:  
    pushl $0  
    pushl $0  
    jmp alltraps  
  
.globl vector1  
  
vector1:  
    pushl $0  
    pushl $1  
  
    jmp alltraps
```

trapasm.S

```
#include "mmu.h"  
  
# vectors.S sends all traps  
here.  
  
.globl alltraps  
  
alltraps:  
    # Build trap frame.  
    pushl %ds  
    pushl %es  
    pushl %fs  
    pushl %gs  
    Pushal  
  
    ....
```

How will interrupts be handled?

On int instruction/interrupt the CPU does this:

- Fetch the n'th descriptor from the IDT, where n is the argument of int. (IDTR->idt[n])
- Check that CPL in %cs is <= DPL, where DPL is the privilege level in the descriptor.
 - Temporarily save %esp and %ss in CPU-internal registers, but only if the target segment selector's PL < CPL.
 - Switching from user mode to kernel mode. Hence save user code's SS and ESP
- Load %ss and %esp from a task segment descriptor.
 - Stack changes to kernel stack now. TS descriptor is on GDT, index given by TR register. See switchuvm()
- Push %ss. // optional
- Push %esp. // optional (also changes ss,esp using TSS)
- Push %eflags.
- Push %cs.
- Push %eip.
- Clear the IF bit in %eflags, but only on an interrupt.
- Set %cs and %eip to the values in the descriptor.

After “int” ‘s job is done

- **IDT was already set**
 - Remember vectors.S
- **So jump to 64th entry in vector’s vector64:**

```
pushl $0
pushl $64
jmp alltraps
```
- **So now stack has ss, esp,eflags, cs, eip, 0 (for error code), 64**
- **Next run alltraps from trapasm.S**

```
# Build trap frame.  
pushl %ds  
pushl %es  
pushl %fs  
pushl %gs  
pushal // push all gen purpose  
regs  
# Set up data segments.  
movw $(SEG_KDATA<<3), %ax  
movw %ax, %ds  
movw %ax, %es  
# Call trap(tf), where tf=%esp  
pushl %esp # first arg to trap()  
call trap  
addl $4, %esp
```

alltraps:

- Now stack contains
- **ss, esp,eflags, cs, eip, 0 (for error code), 64, ds, es, fs, gs, eax, ecx, edx, ebx, oesp, ebp, esi, edi**
 - This is the struct trapframe !
 - So the kernel stack now contains the trapframe
 - Trapframe is a part of kernel stack

```
void  
trap(struct trapframe *tf)  
{  
    if(tf->trapno == T_SYSCALL){  
        if(myproc()->killed)  
            exit();  
        myproc()->tf = tf;  
        syscall();  
        if(myproc()->killed)  
            exit();  
        return;  
    }  
    switch(tf->trapno){  
        ....
```

trap()

- Argument is trapframe
- In alltraps
 - Before “call trap”, there was “push %esp” and stack had the trapframe
 - Remember calling convention --> when a function is called, the stack contains the arguments in reverse order (here only 1 arg)

trap()

- **Has a switch**
 - `switch(tf->trapno)`
 - Q: who set this trapno?
- **Depending on the type of trap**
 - Call interrupt handler
- Timer
 - `wakeup(&ticks)`
- IDE: disk interrupt
 - `Ideintr()`
- KBD
 - `KbdINTR()`
- COM1
 - `UatINTR()`
- If Timer
 - Call `yield()` -- calls `sched()`
- If process was killed (how is that done?)
 - Call `exit()`!

when trap() returns

- #Back in alltraps

```
call trap
```

```
addl $4, %esp
```

```
# Return falls through to trapret...
```

```
.globl trapret
```

```
trapret:
```

```
popal
```

```
popl %gs
```

```
popl %fs
```

```
popl %es
```

```
popl %ds
```

```
addl $0x8, %esp # trapno and errcode
```

```
iret
```

```
.
```

- Stack had (trapframe)
 - ss, esp,eflags, cs, eip, 0 (for error code), 64, ds, es, fs, gs, eax, ecx, edx, ebx, oesp, ebp, esi, edi, esp
- add \$4 %esp
 - esp
- popal
 - eax, ecx, edx, ebx, oesp, ebp, esi, edi
- Then gs, fs, es, ds
- add \$0x8, %esp
 - 0 (for error code), 64
- iret
 - ss, esp,eflags, cs, eip,

Scheduler

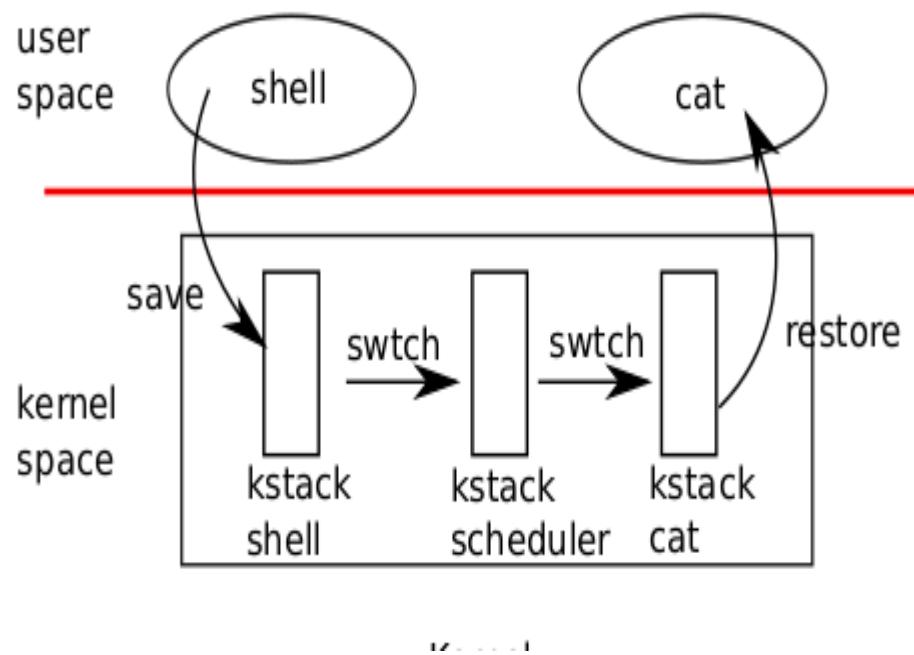
Scheduler – in most simple terms

- **Selects a process to execute and passes control to it !**
 - The process is chosen out of “READY” state processes
 - Saving of context of “earlier” process and loading of context of “next” process needs to happen
- **Questions**
 - What are the different scenarios in which a scheduler called ?
 - What are the intricacies of “passing control”
 - What is “context” ?

Steps in scheduling scheduling

- Suppose you want to switch from P1 to P2 on a timer interrupt
- P1 was doing
 - F() { i++; j++; }
- P2 was doing
 - G() { x--; y++; }
- P1 will experience a timer interrupt, switch to kernel (scheduler) and scheduler will schedule P2

4 stacks need to change!



- **User stack of process -> kernel stack of process**
 - Switch to kernel stack
 - The normal sequence on any interrupt !
- **Kernel stack of process -> kernel stack of scheduler**
 - Why?
- **Kernel stack of scheduler -> kernel stack of new process . Why?**
- **Kernel stack of new process -> user stack of new process**

scheduler()

- **Enable interrupts**
- **Find a RUNNABLE process. Simple round-robin!**
- **c->proc = p**
- **switchuvm(p) : Save TSS and make CR3 to point to new process pagedir**
- **p->state = RUNNING**
- **swtch(&(c->scheduler), p->context)**

swtch

swtch:

```
movl 4(%esp), %eax
```

```
movl 8(%esp), %edx
```

```
# Save old callee-saved registers
```

```
pushl %ebp
```

```
pushl %ebx
```

```
pushl %esi
```

```
pushl %edi
```

```
# Switch stacks
```

```
movl %esp, (%eax)
```

```
movl %edx, %esp
```

```
# Load new callee-saved registers
```

```
popl %edi
```

```
popl %esi
```

```
popl %ebx
```

```
popl %ebp
```

```
ret
```

scheduler()

- **swtch(&(c->scheduler), p->context)**
- **Note that when scheduler() was called, when P1 was running**
- **After call to swtch() shown above**
 - The call does NOT return!
 - The new process P2 given by ‘p’ starts running !
 - Let’s review swtch() again

swtch(old, new)

- The magic function in swtch.S
- Saves callee-save registers of old context
- Switches esp to new-context's stack
- Pop callee-save registers from new context

ret

- where? in the case of first process – returns to forkret() because stack was setup like that !
- in case of other processes, return where?
 - Return address given on kernel stack. But what's that?
 - The EIP in p->context
 - When was EIP set in p->context ?

scheduler()

- **Called from?**
 - `mpmain()`
 - **No where else!**
- **`sched()` is another scheduler function !**
 - **Who calls `sched()` ?**
 - `exit()` - a process exiting calls `sched ()`
 - `yield()` - a process interrupted by timer calls `yield()`
 - `sleep()` - a process going to wait calls `sleep()`

sched()

```
void
sched(void)
{
    int intena;
    struct proc *p = myproc();
    if(!holding(&ptable.lock))
        panic("sched ptable.lock");
    if(mycpu()->ncli != 1)
        panic("sched locks");
    if(p->state == RUNNING)
        panic("sched running");
    if(readeflags()&FL_IF)
        panic("sched interruptible");
    intena = mycpu()->intena;
    swtch(&p->context, mycpu()->scheduler);
/*A*/ mycpu()->intena = intena;
}
```

- get current process
- Error checking code (ignore as of now)
- get interrupt enabled status on current CPU (ignore as of now)
- call to swtch
 - Note the arguments' order
 - p->context first, mycpu()->scheduler second
- swtch() is a function call
 - pushes address of /*A*/ on stack of current process p
 - switches stack to mycpu()->scheduler. Then pops EIP from that stack and jumps there.
 - when was mycpu()->scheduler set? Ans: during scheduler()

sched() and scheduler()

```
 sched() {  
 ...  
     swtch(&p->context, mycpu()->scheduler); /* X */  
 }  
 }
```

```
 scheduler(void) {  
 ...  
     swtch(&(c->scheduler), p->context); /* Y */  
 }
```

- **scheduler() saves context in c->scheduler, sched() saves context in p->context**
- **after swtch() call in sched(), the control jumps to Y in scheduler**
 - **Switch from process stack to scheduler's stack**
- **after swtch() call in scheduler(), the control jumps to X in sched()**
 - **Switch from scheduler's stack to new process's stack**
- **Set of co-operating functions**

sched() and scheduler() as co-routines

- **In sched()**

```
swtch(&p->context, mycpu()->scheduler);
```

- **In scheduler()**

```
swtch(&(c->scheduler), p->context);
```

- **These two keep switching between processes**
- **These two functions work together to achieve scheduling**
- **Using asynchronous jumps**
- **Hence they are co-routines**

To summarize

- **On a timer interrupt during P1**
 - trap() is called. **Stack has changed from P1's user stack to P1's kernel stack**
 - trap()->yield()
 - yield()->sched()
 - sched() -> swtch(&p->context, c->scheduler())
 - **Stack changes to scheduler's kernel stack.**
 - **Switches to location “Y” in scheduler().**
- **Now the loop in scheduler()**
 - calls switchkvm()
 - Then continues to find next process (P2) to run
 - Then calls switchuvvm(p): changing the page table to the P2's page tables
 - then calls swtch(&c->scheduler, p2->context)
 - **Stack changes to P2's kernel stack.**
 - P2 runs the last instruction it was was in ! Where was it?
 - mycpu()->intena = intena; in sched()
 - Then returns to the one who called sched() i.e. exit/sleep, etc
 - Finally returns from it's own “TRAP” handler and **returns to P2's user stack and user code**

File Systems

Abhijit A M
abhijit.comp@coep.ac.in

What we are going to learn

- **The operating system interface (system calls, commands/utilities) for accessing files in a file-system**
- **Design aspects of OS to implement the file system**
 - **On disk data structure**
 - **In memory kernel data structures**

What is a file?

- A (dumb!) sequence of bytes (typically on a permanent storage:secondary, tertiary) , with
 - A name
 - Permissions
 - Owner
 - Timestamps,
 - Etc.
- Types: Text files, binary files (one classification)
 - Text: All bytes are human readable
 - Binary: Non-text
- Types: ODT, MP4, TXT, DOCX, etc. (another classification)
 - Most typically describing the organization of the data inside the file
 - Each type serving the needs of a particular application (not kernel)

File types and kernel

- **For example, MP4 file**
 - **vlc will do a open(...) on the file, and call read(...), interpret the contents of the file as movie and show movie**
 - **Kernel will simply provide open(...), read(...), write(...) to access file data**
 - **Meaning of the file contents is known to VLC and not to kernel!**

file type	usual extension	function
executable	exe, com, bin or none	ready-to-run machine-language program
object	obj, o	compiled, machine language, not linked
source code	c, cc, java, pas, asm, a	source code in various languages
batch	bat, sh	commands to the command interpreter
text	txt, doc	textual data, documents
word processor	wp, tex, rtf, doc	various word-processor formats
library	lib, a, so, dll	libraries of routines for programmers
print or view	ps, pdf, jpg	ASCII or binary file in a format for printing or viewing
archive	arc, zip, tar	related files grouped into one file, sometimes compressed, for archiving or storage
multimedia	mpeg, mov, rm, mp3, avi	binary file containing audio or A/V information

What is a file?

- The sequence of bytes can be *interpreted (by an application)* to be
 - Just a sequence of bytes
 - E.g. a text file
 - Sequence of records/structures
 - E.g. a file of student records , by database application, etc
 - A complexly organized, collection of records and bytes
 - E.g. a “ODT” or “DOCX” file
- What's the role of OS in above mentioned file type, and organization?
 - Mostly NO role on Unixes, Linuxes!
 - They are handled by applications !
 - Types handled by OS: normal file, directory, block device file, character device file, FIFO file (named pipe), etc.
 - Also types handled by OS: executable file, non-executable file

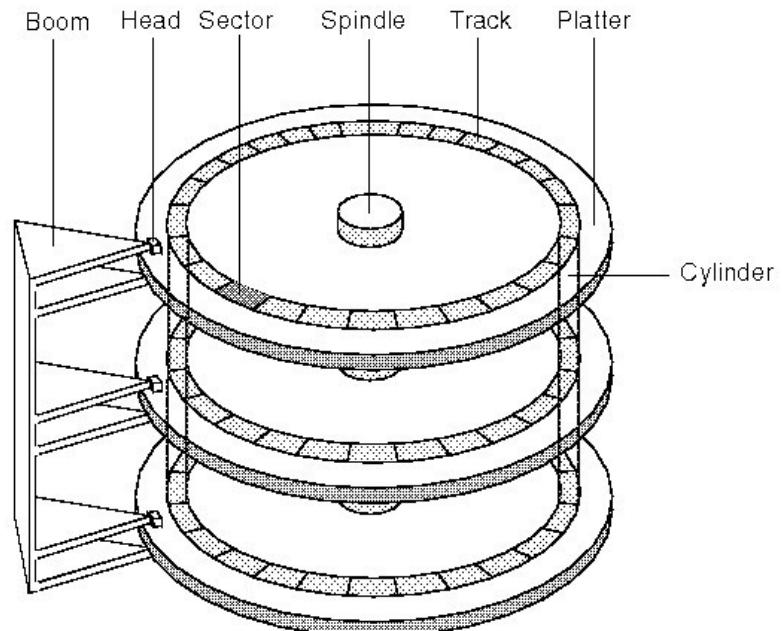
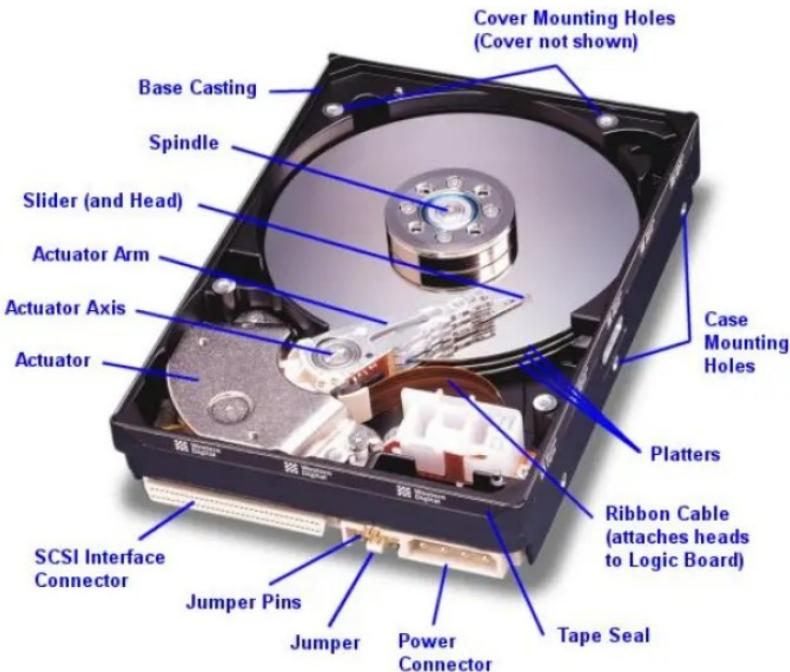
File attributes

- Run
 - \$ ls -l
 - on Linux
 - To see file listing with different attributes
- Different OSes and file-systems provide different sets of file attributes
 - Some attributes are common to most, while some are different
 - E.g. name, size, owner can be found on most systems
 - “Executable” permission may not be found on all systems

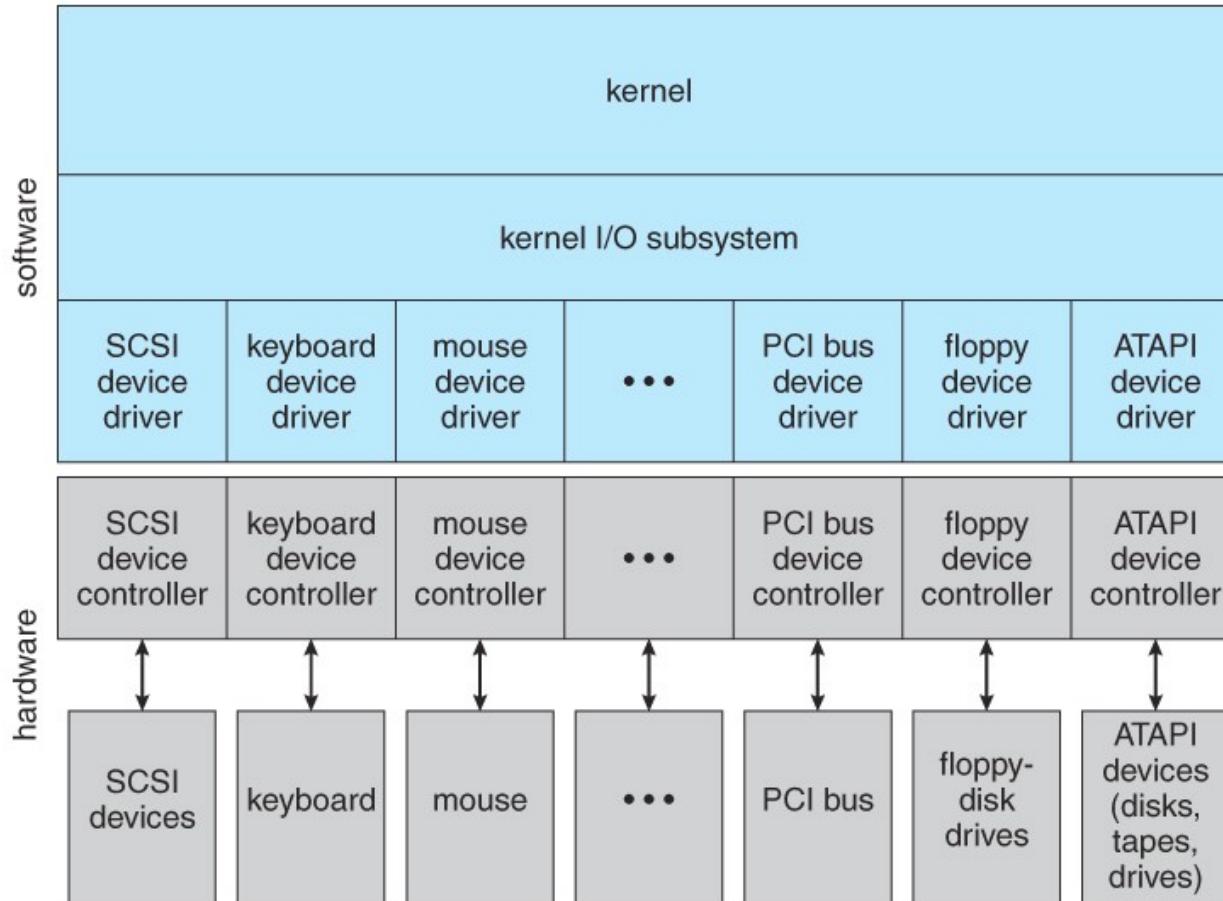
Access methods

- OS system calls may provide two types of access to files
 - Sequential Access
 - **read next**
 - **write next**
 - **reset**
 - **no read after last write (rewrite)**
 - Linux provides sequential access using **open()**, **read()**, **write()**, ...
 - Direct Access
 - **read n**
 - **write n**
 - **position to n**
 - read next**
 - write next**
 - **rewrite n**
- n = relative block number**
- **pread(), pwrite() on Linux**
 - **ssize_t pread(int fd, void *buf, size_t count, off_t offset);**
 - **ssize_t pwrite(int fd, const void *buf, size_t count, off_t offset);**

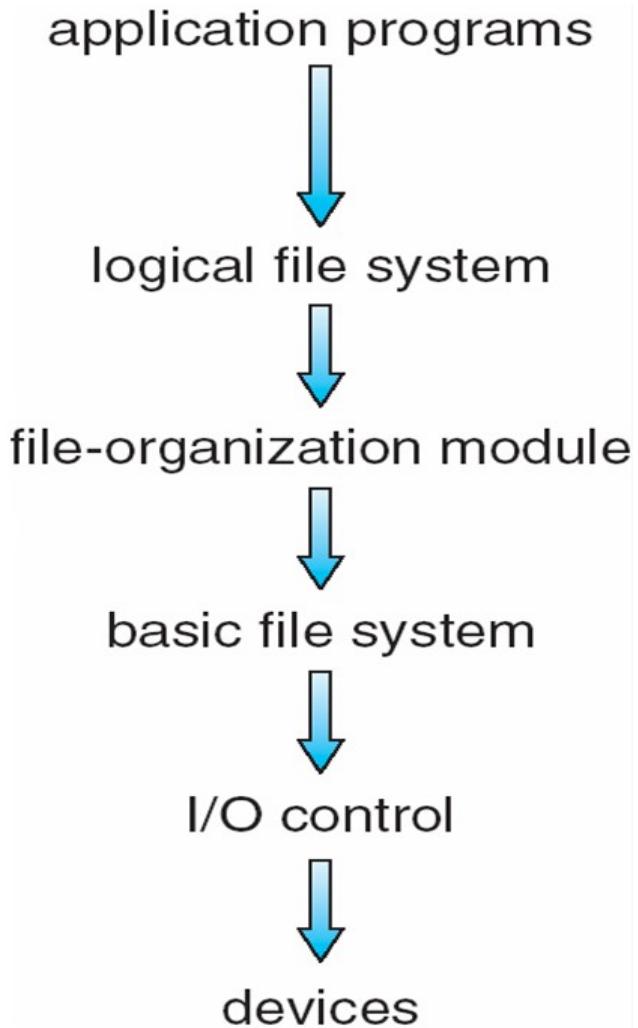
Disk



Device Driver



File system implementation: layering



Application programs

```
int main() {  
    char buf[128]; int count;  
    fd = open(...);  
    read(fd, buf, count);  
}  
-----
```

OS

Logical file system:

```
sys_read(int fd, char *buf, int count) {  
    file *fp = currproc->fdarray[fd];  
    file_read(fp, ...);  
}
```

File organization module:

```
file_read(file *fp, char *buf, int count) {  
    offset = fp->current_offset;  
    translate offset into blockno;  
    basic_read(blockno, buf, count);  
}
```

Basic File system:

```
basic_read(int blockno, char *buf, ...) {  
    os_buffer *bp;  
    sectorno = calculation on blockno;  
    disk_driver_read(sectorno, bp );  
    move-process-to-wait-queue;  
    copy-data-to-user-buffer(bp, buf);  
}
```

IO Control, Device driver:

```
disk_driver_read(sectorno) {  
    issue instructions to disk controller  
(often assembly code)  
    to read sectorno into specific  
location;  
}
```

XV6 does it slightly differently, but following the layering principle!

OS's job now

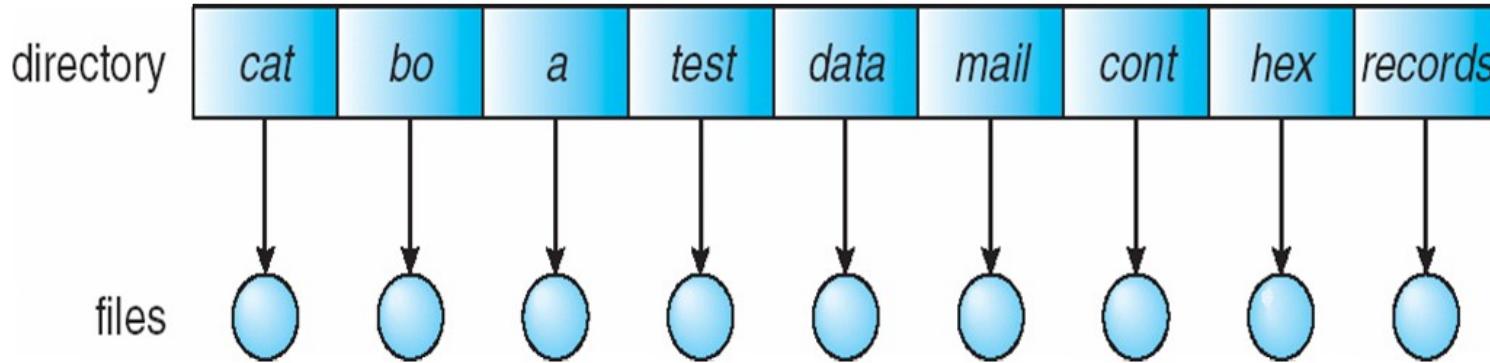
- To implement the logical view of file system as seen by end user
- Using the logical block-based view offered by the device driver

Formatting

- **Physical hard disk divided into partitions**
 - Partitions also known as minidisks, slices
- **A raw disk partition is accessible using device driver – but no block contains any data !**
 - Like an un-initialized array, or sectors/blocks
- **Formatting**
 - Creating an initialized data structure on the partition, so that it can start storing the acyclic graph tree structure on it
 - Different formats depending on different implementations of the directory tree structure: ext4, NTFS, vfat, VxFS, ReiserFS, WafleFS, etc.
- **Formatting happens on “a physical partition” or “a logical volume made available by volume manager”**

Different types of “layouts”

Single level directory



Naming problem

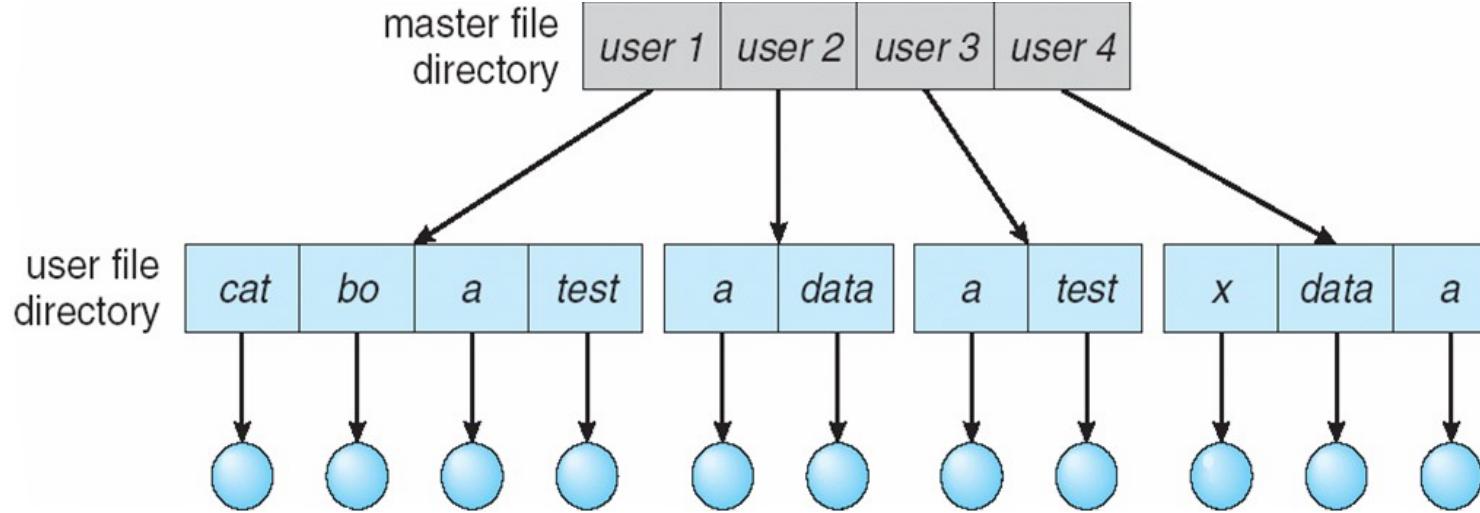
Grouping problem

Example: RT-11, from 1970s

<https://en.wikipedia.org/wiki/RT-11>

Different types of “layouts”

Two level directory



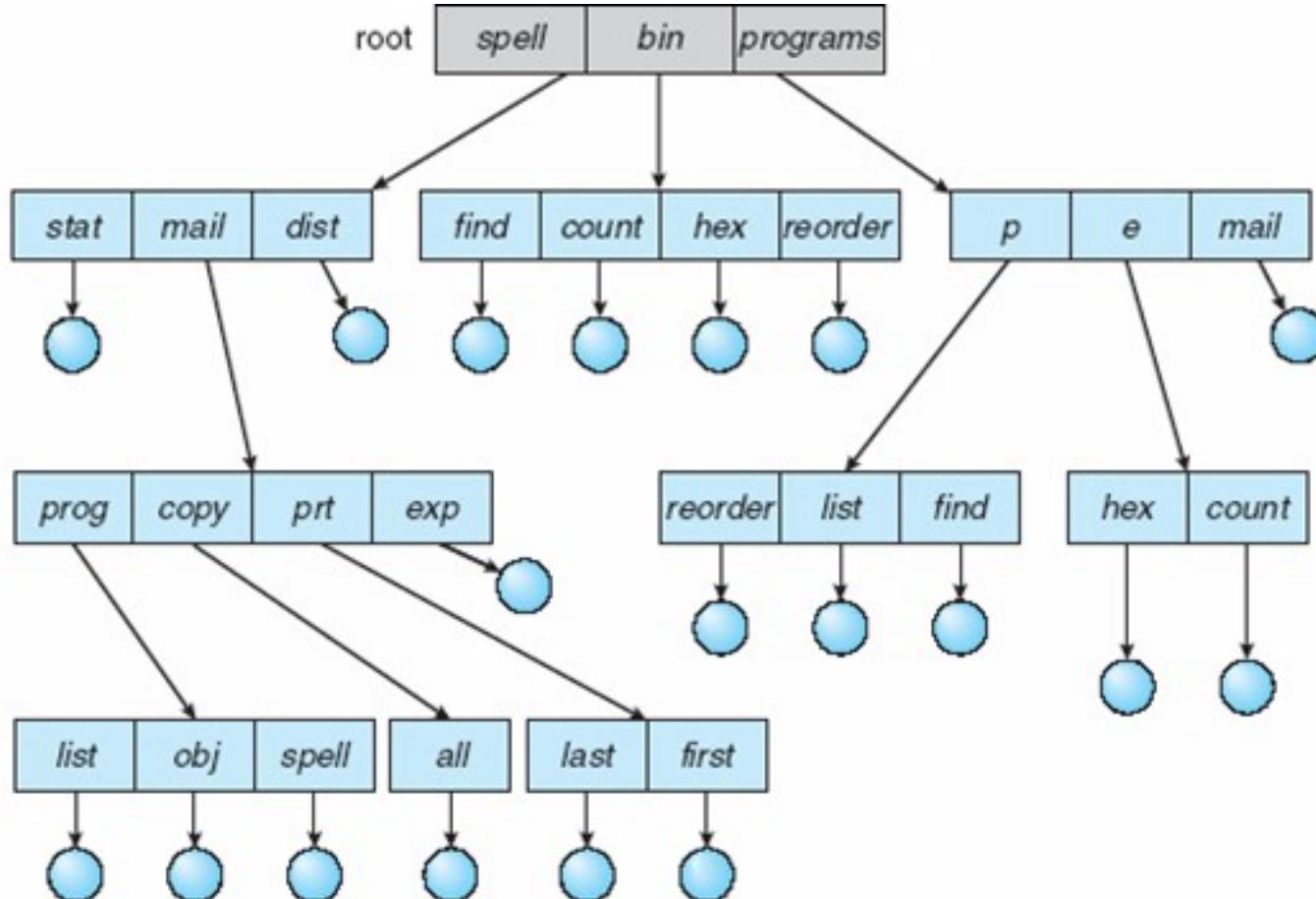
Path name

Can have the same file name for different user

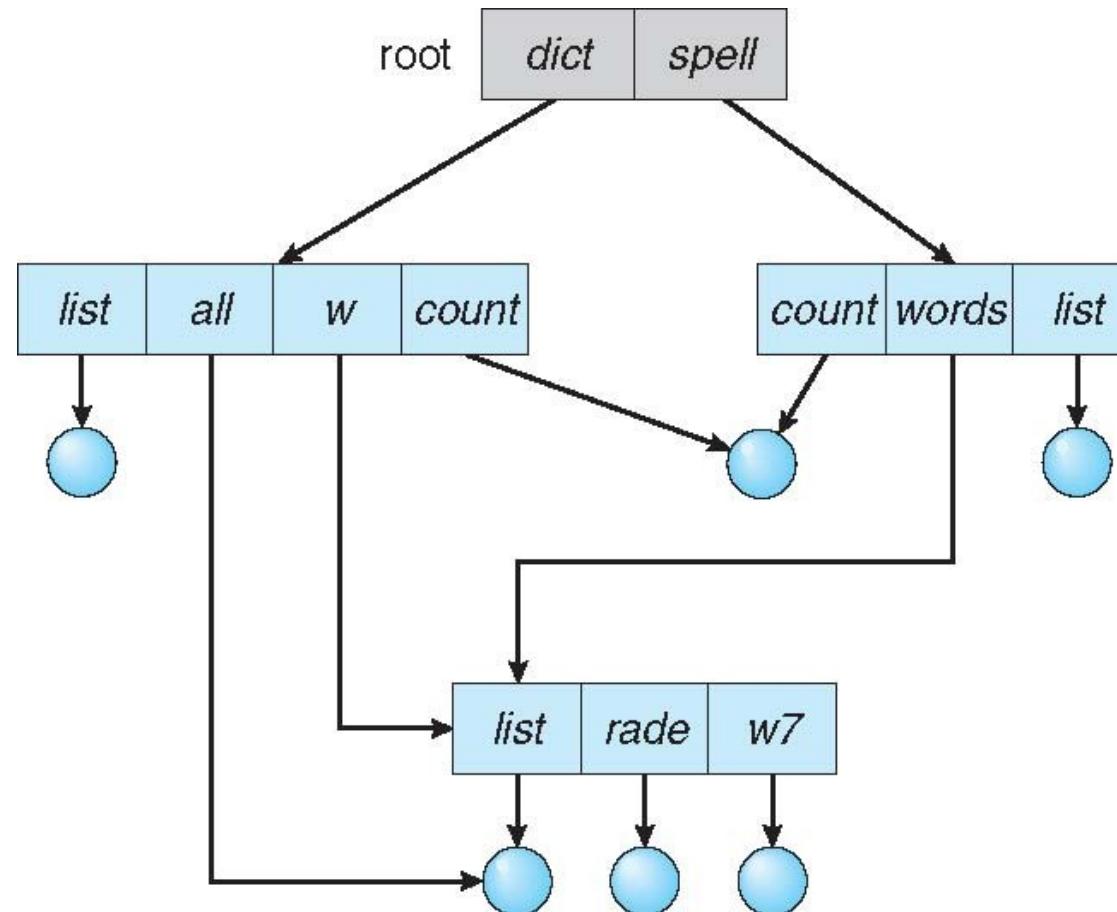
Efficient searching

No grouping capability

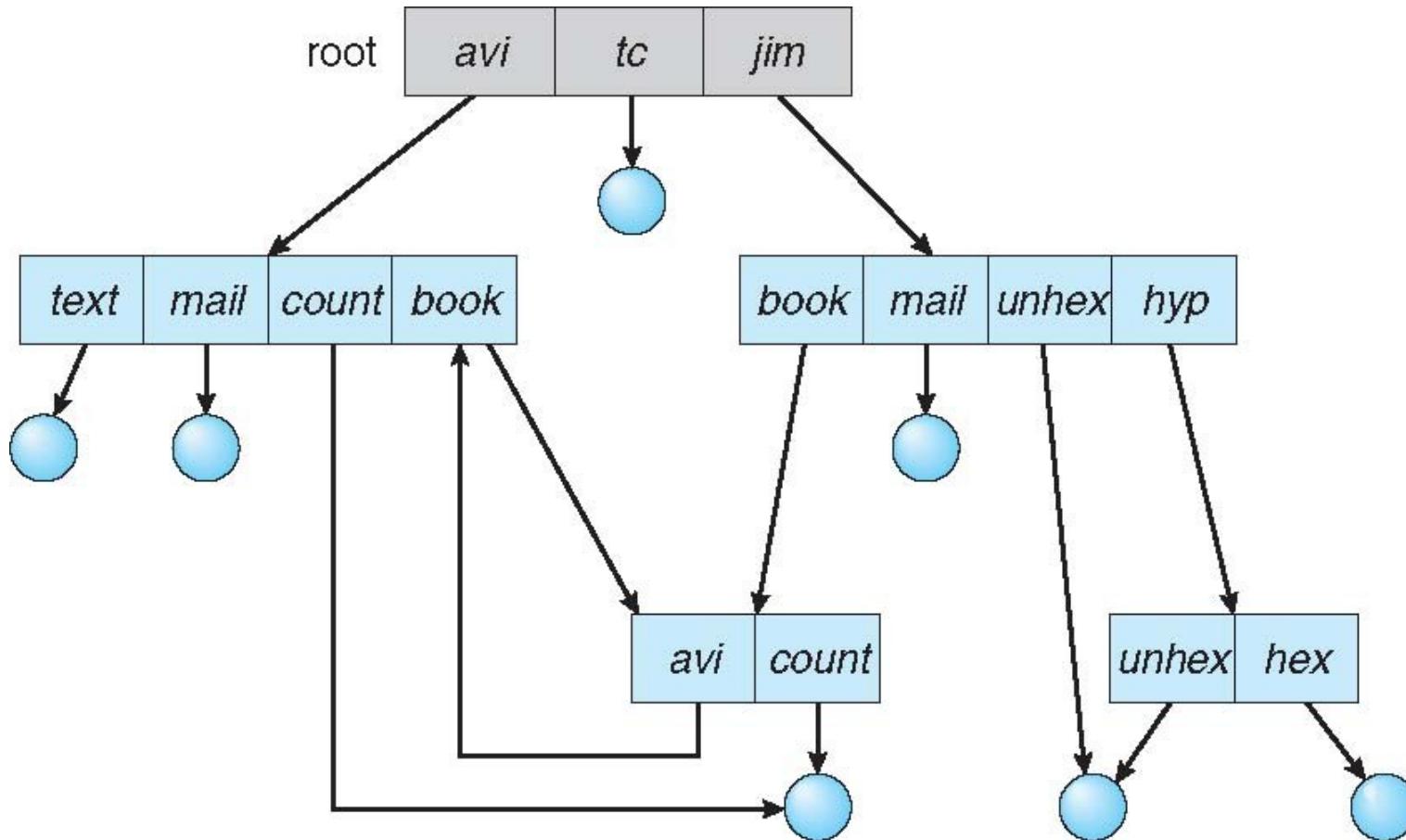
Tree Structured directories



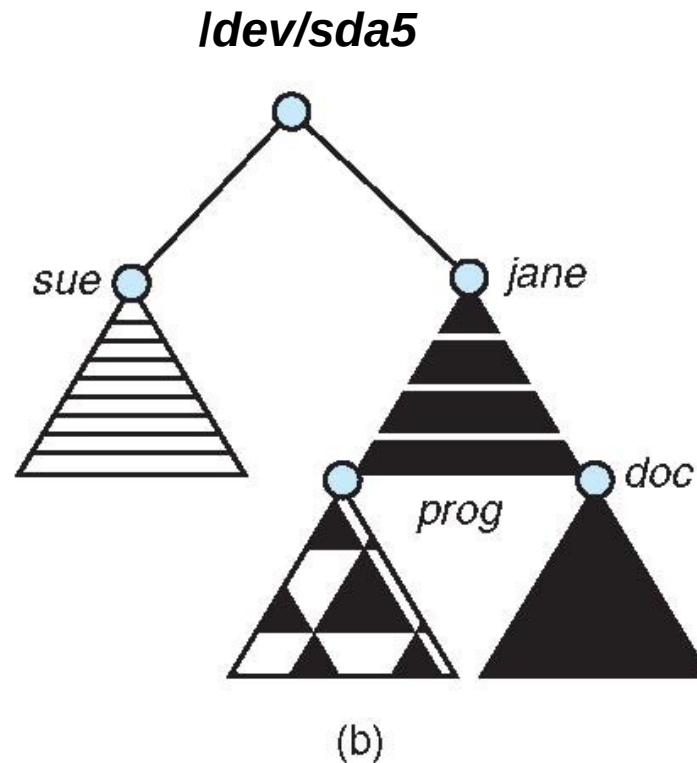
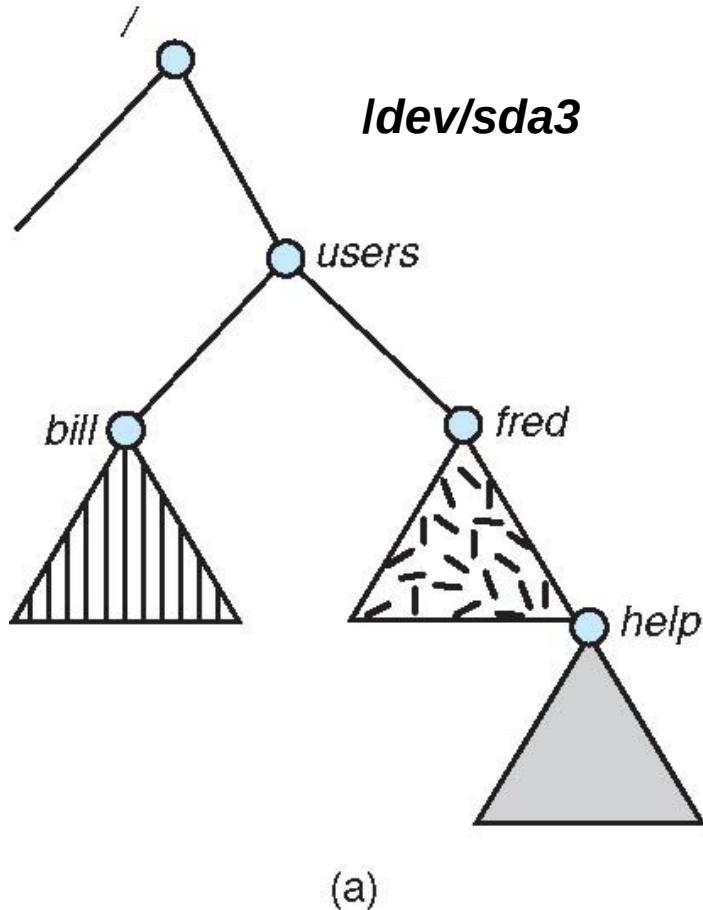
Acyclic Graph Directories



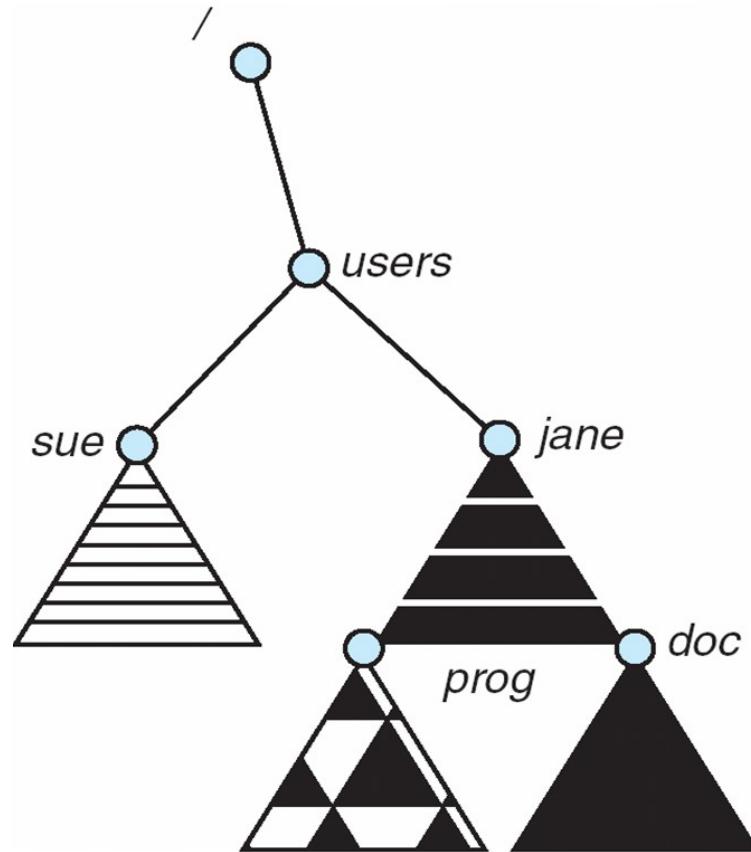
General Graph directory



Mounting of a file system: before



Mounting of a file system: after



`$sudo mount /dev/sda5 /users`

Remote mounting: NFS

- Network file system
- **\$ sudo mount 10.2.1.2:/x/y /a/b**
 - The */x/y* partition on 10.2.1.2 will be made available under the folder */a/b* on this computer
-

File sharing semantics

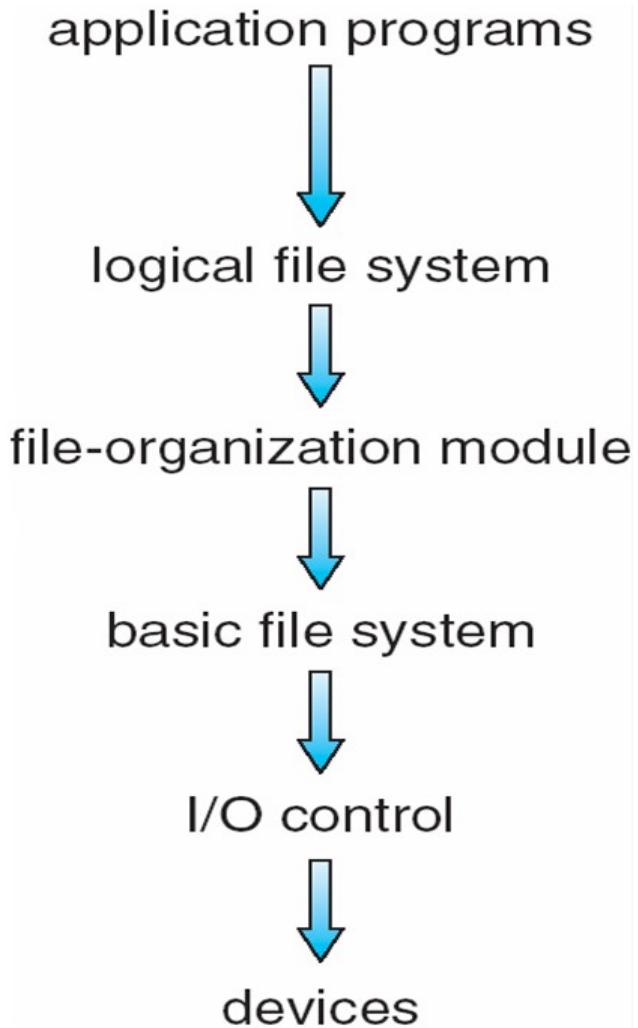
- **Consistency semantics specify how multiple users are to access a shared file simultaneously**
- **Unix file system (UFS) implements:**
 - Writes to an open file visible immediately to other users of the same open file
 - One mode of sharing file pointer to allow multiple users to read and write concurrently
- **AFS has session semantics**
 - Writes only visible to sessions starting after the file is closed

Implementing file systems

File system on disk

- **What we know**
 - Disk I/O in terms of sectors (512 bytes)
 - File system: implementation of acyclic graph using the linear sequence of sectors
 - Store a acyclic graph into array of “blocks”/“sectors”
 - Device driver available: gives sector/block wise access to the disk

File system implementation: layering



File system: Layering

- Device drivers manage I/O devices at the I/O control layer
 - Given commands like “read drive1, cylinder 72, track 2, sector 10, into memory location 1060” outputs low-level hardware specific commands to hardware controller
- Basic file system given command like “retrieve block 123” translates to device driver
 - Also manages memory buffers and caches (allocation, freeing, replacement)
 - Buffers hold data in transit
 - Caches hold frequently used data
- File organization module understands files, logical address, and physical blocks
 - Translates logical block # to physical block #
 - Manages free space, disk allocation
- Logical file system manages metadata information
 - Translates file name into file number, file handle, location by maintaining file control blocks (inodes in Unix)
 - Directory management
 - Protection

File system implementation: Different problems to be solved

- **What to do at boot time, how to locate kernel ?**
- **How to store directories and files on the partition ?**
 - Complex problem. Hierarchy + storage allocation + efficiency + limits on file/directory sizes + links (hard, soft)
- **How to manage list of free sectors/blocks?**
- **How to store the summary information about the complete file system : #files, #free-blocks, ...**
- **How to mount a file system , how to unmount?**

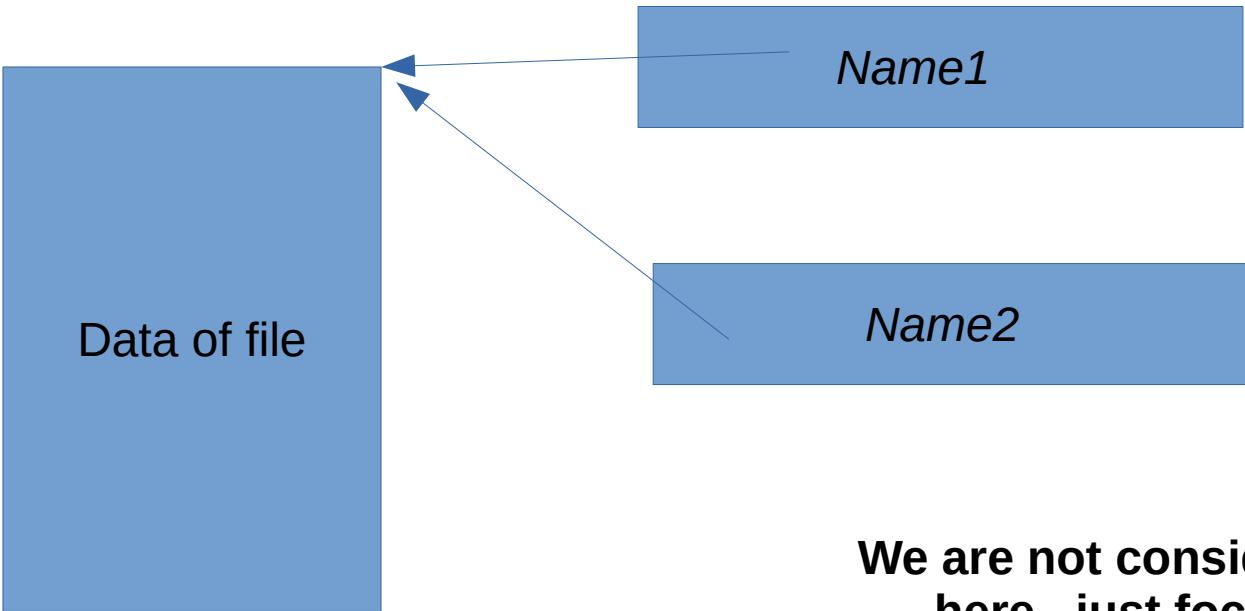
File system implementation: Different problems to be solved

- **About storing a file, how to store**
 - Data
 - Attributes
 - Name
 - Link count

The hard link problem

- Need to separate name from data !
 - */x/y and /a/b* should be same file. How?
 - Both names should refer to same data !
 - Data is separated separately from name, and the name gives a “reference” to data
- What about attributes ?
 - They go with data! (not with name!)
- So solution was: indirection !

The hard link problem



**We are not considering other problems
here , just focussing on hard link
problem**

A typical file control block (inode)

file permissions

Name is stored
separately

file dates (create, access, write)

Where?

file owner, group, ACL

IN data block of
directory

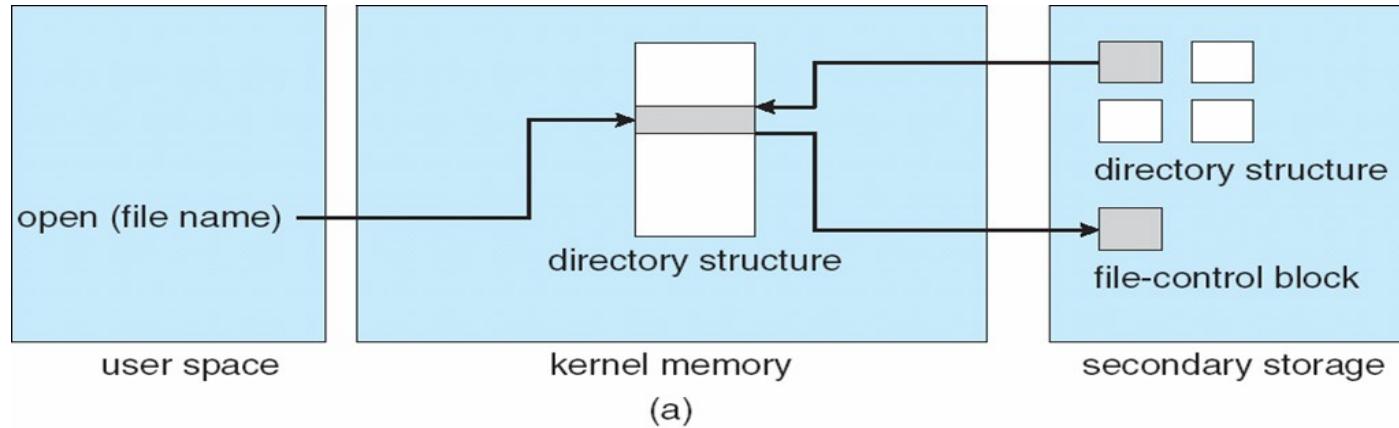
file size

file data blocks or pointers to file data blocks

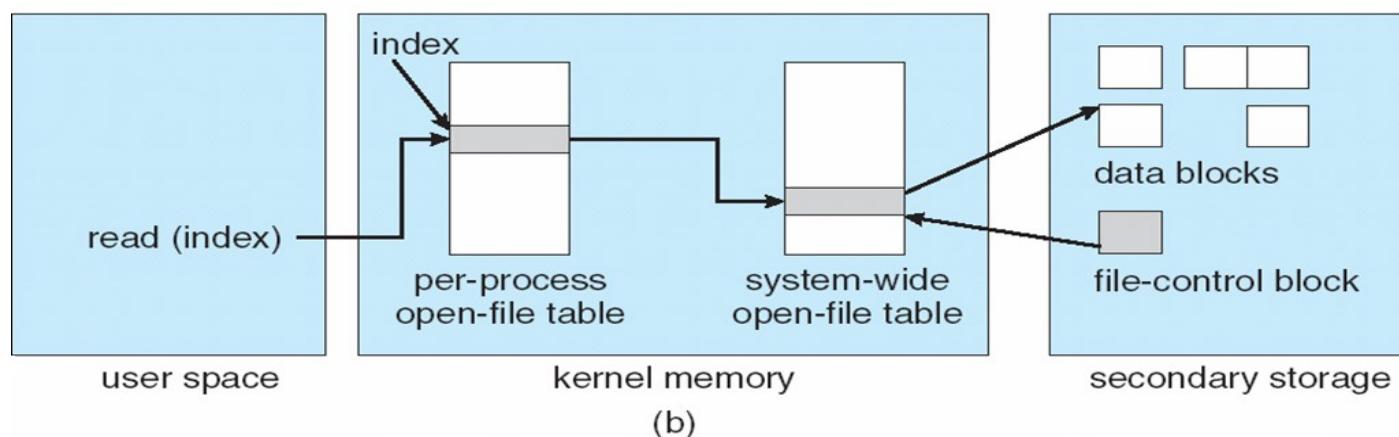
In memory data structures

- **Mount table**
 - storing file system mounts, mount points, file system types
- **See next slide for “file” related data structures**
- **Buffers**
 - hold data blocks from secondary storage

In memory data structures: for open,read,write, ...



Open returns a file handle for subsequent use



Data from read eventually copied to specified user process memory address

At boot time

- **Root partition**
 - Contains the file system hosting OS
 - “mounted” at boot time – contains “/”
 - Normally can’t be unmounted!
- **Check all other partitions**
 - Specified in `/etc/fstab` on Linux
 - Check if the data structure on them is consistent
 - Consistent != perfect/accurate/complete

Directory Implementation

- **Problem**
 - Directory contains files and/or subdirectories
 - Operations required – create files/directories, access files/directories, search for a file (during lookup), etc.
 - Directory needs to give location of each file on disk

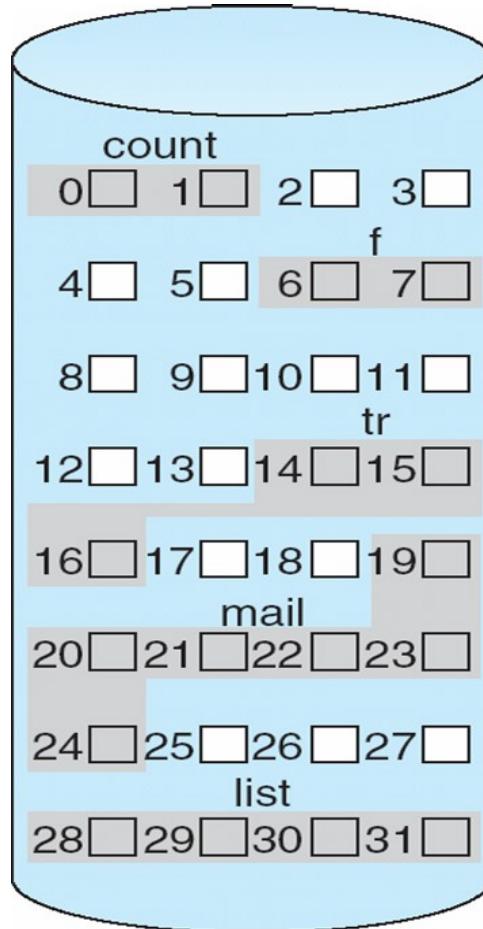
Directory Implementation

- **Linear list of file names with pointer to the data blocks**
 - Simple to program
 - Time-consuming to execute
 - Linear search time
 - Could keep ordered alphabetically via linked list or use B+ tree
 - Ext2 improves upon this approach.
- **Hash Table – linear list with hash data structure**
 - Decreases directory search time
 - Collisions – situations where two file names hash to the same location
 - Only good if entries are fixed size, or use chained-overflow method

Disk space allocation for files

- File contain data and need disk blocks/sectors for storing it
- File system layer does the allocation of blocks on disk to files
- Files need to
 - Be created, expanded, deleted, shrunk, etc.
 - How to accommodate these requirements?

Contiguous Allocation of Disk Space



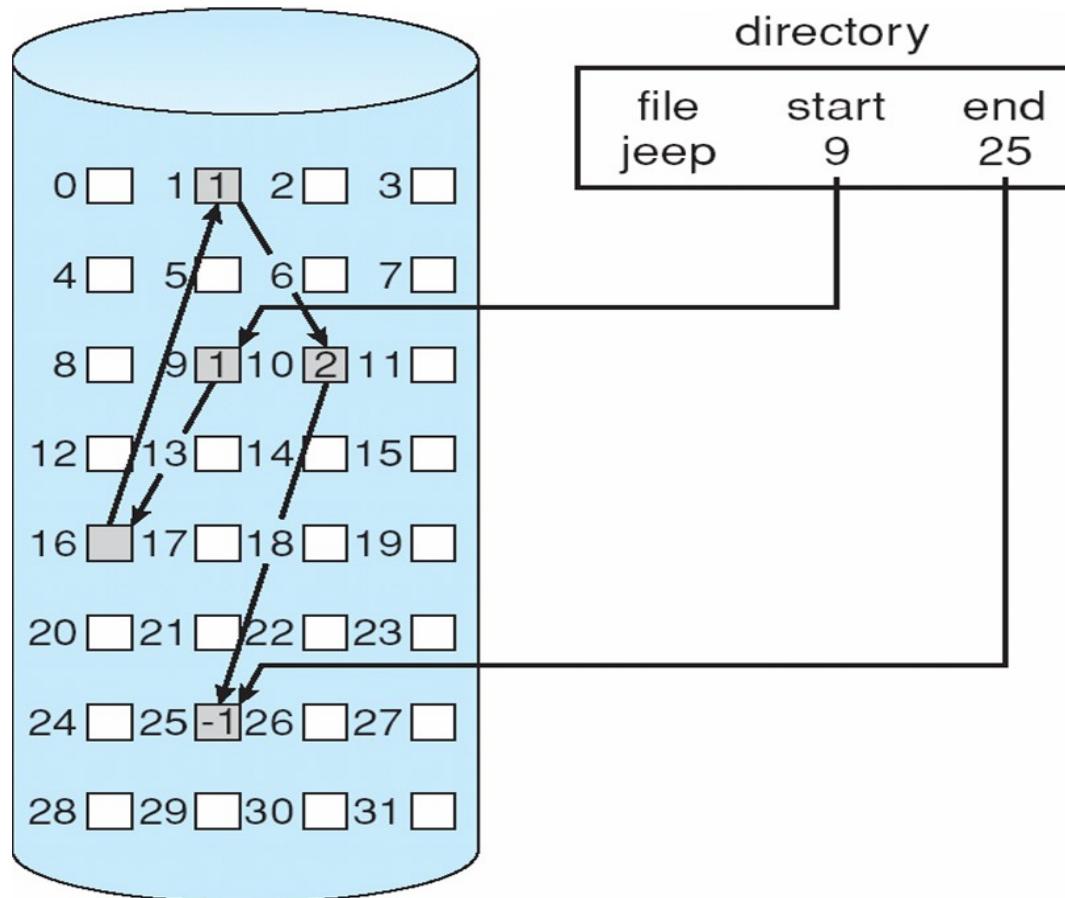
directory

file	start	length
count	0	2
tr	14	3
mail	19	6
list	28	4
f	6	2

Contiguous allocation

- **Each file occupies set of contiguous blocks**
- **Best performance in most cases**
- **Simple – only starting location (block #) and length (number of blocks) are required**
- **Problems include finding space for file, knowing file size, external fragmentation, need for compaction off-line (downtime) or on-line**

Linked allocation of blocks to a file

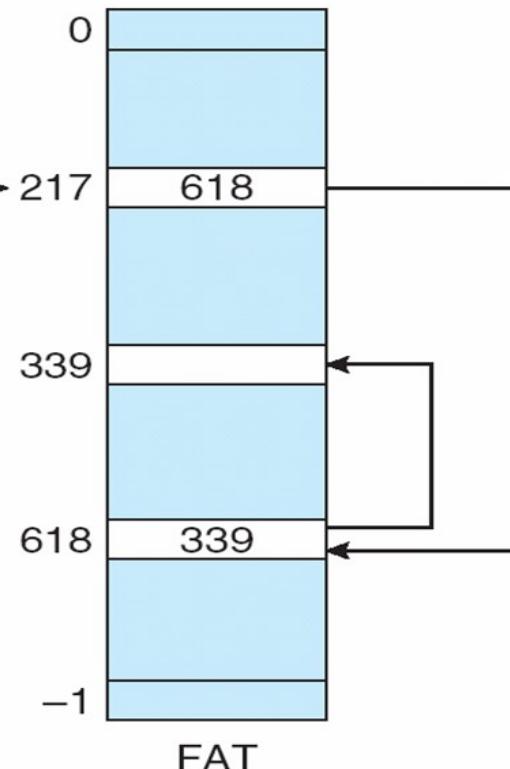
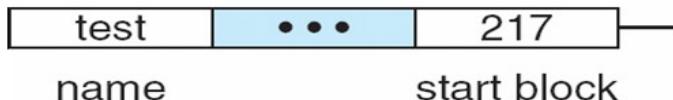


Linked allocation of blocks to a file

- **Linked allocation**
 - Each file a linked list of blocks
 - File ends at nil pointer
 - No external fragmentation
 - Each block contains pointer to next block (i.e. data + pointer to next block)
 - No compaction, external fragmentation
- Free space management system called when new block needed
- Improve efficiency by clustering blocks into groups but increases internal fragmentation
- Reliability can be a problem
- Locating a block can take many I/Os and disk seeks

FAT: File Allocation Table

directory entry



- FAT (File Allocation Table), a variation
 - Beginning of volume has table, indexed by block number
 - Much like a linked list, but faster on disk and cacheable
 - New block allocation simple

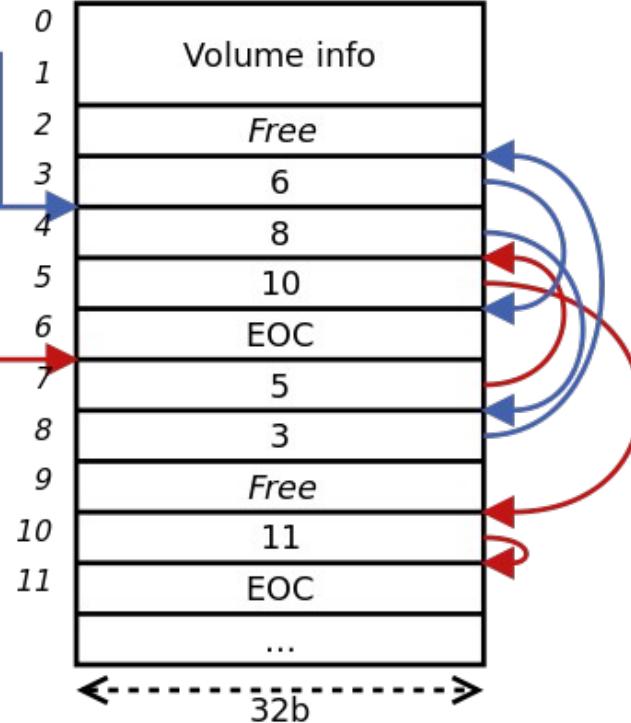
FAT: File Allocation Table

Variants: FAT8,
FAT12, FAT16,
FAT32, VFAT, ...

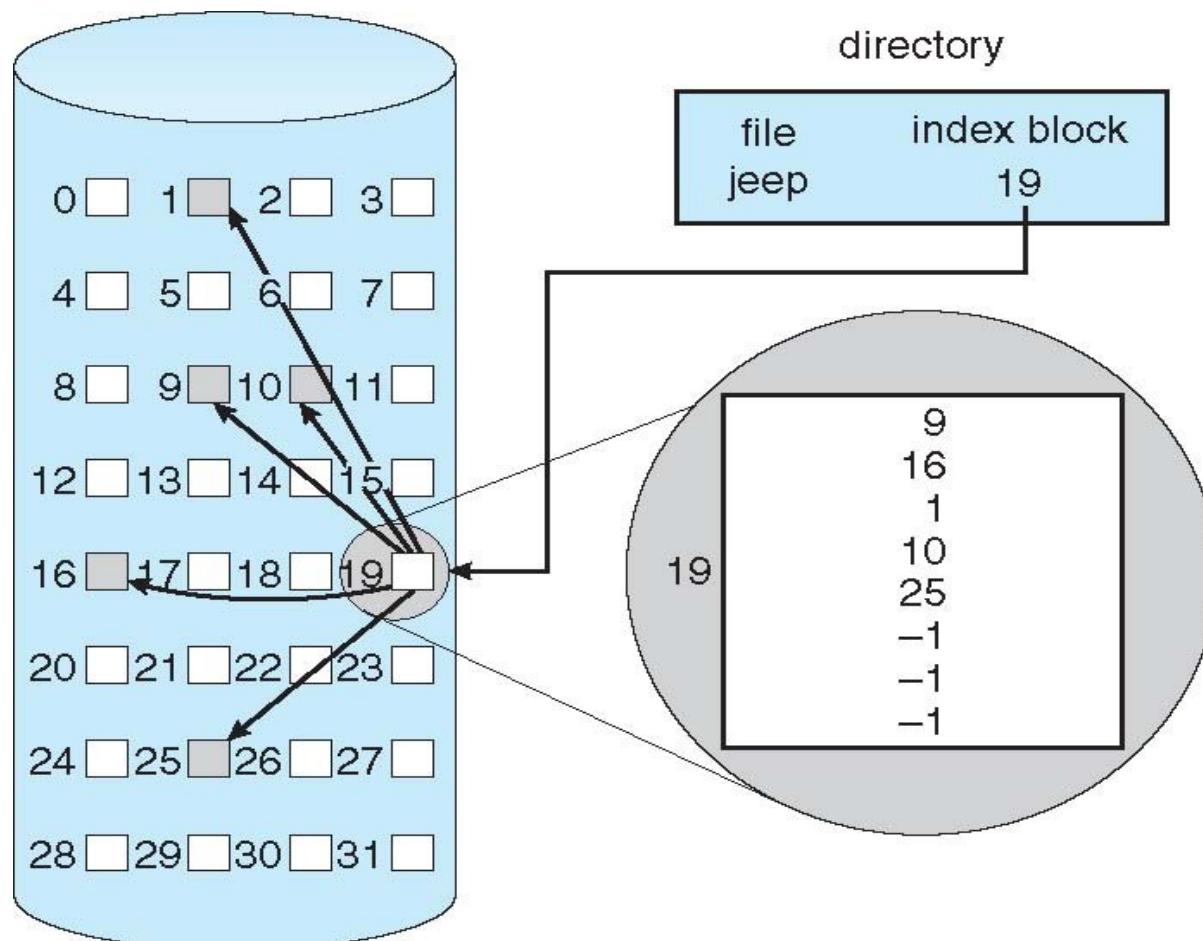
Directory table entry (32B)

Filename (8B)
Extension (3B)
Attributes (1B)
Reserved (1B)
Create time (3B)
Create date (2B)
Last access date (2B)
First cluster # (MSB, 2B)
Last mod. time (2B)
Last mod. date (2B)
First cluster # (LSB, 2B)
File size (4B)

File allocation table



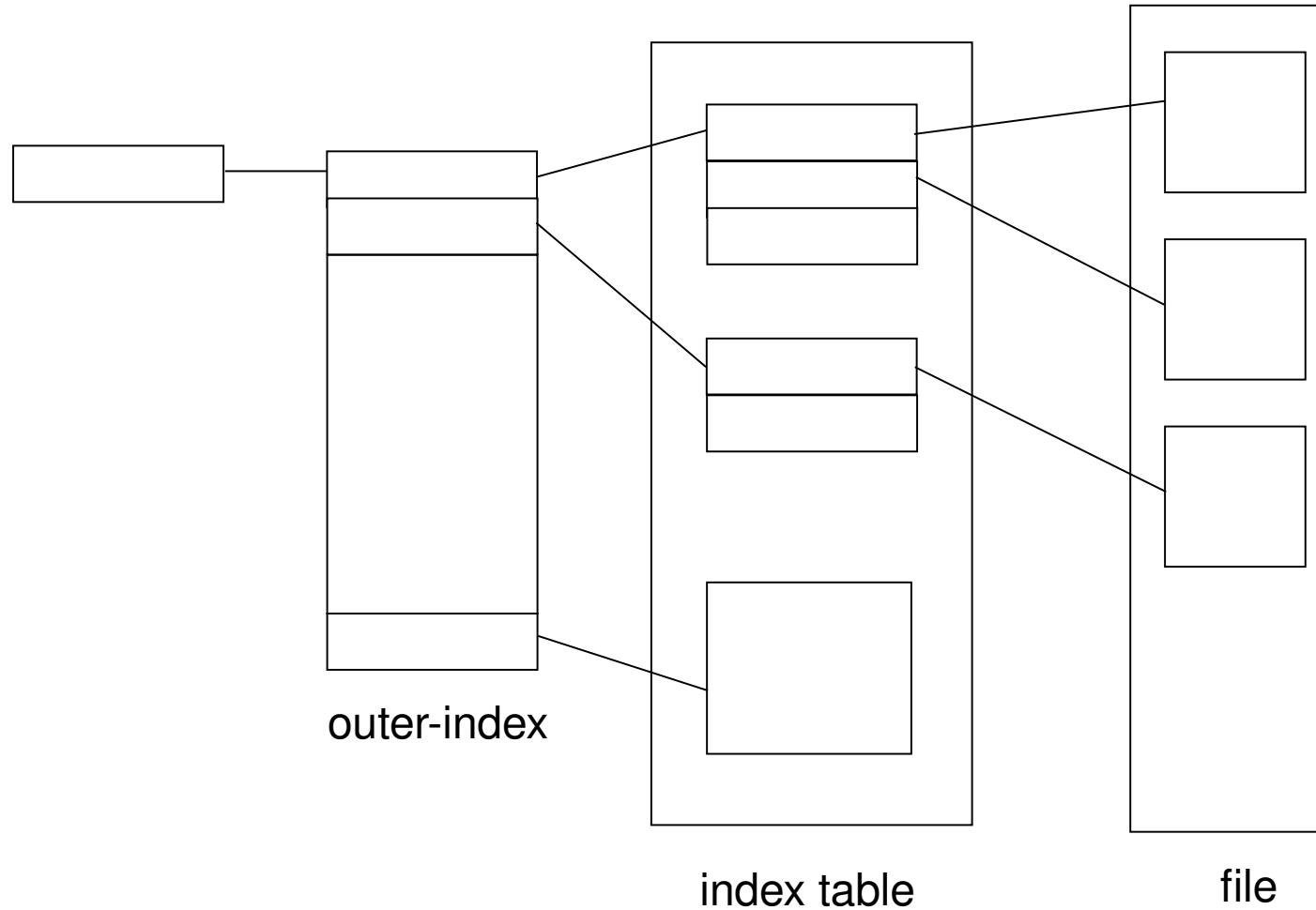
Indexed allocation



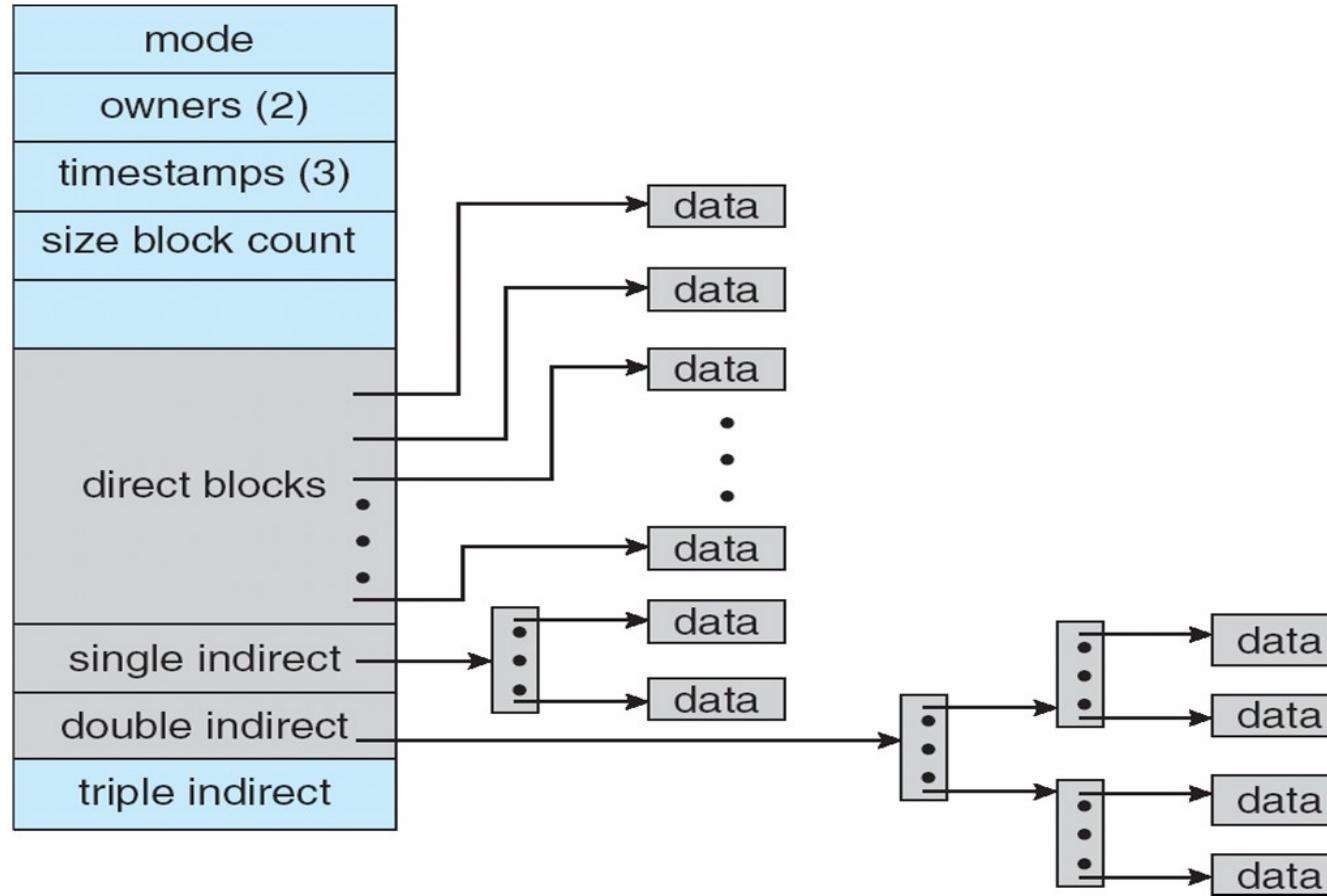
Indexed allocation

- Need index table
- Random access
- Dynamic access without external fragmentation, but have overhead of index block
- Mapping from logical to physical in a file of maximum size of 256K bytes and block size of 512 bytes. We need only 1 block for index table

Multi level indexing



Unix UFS: combined scheme for block allocation



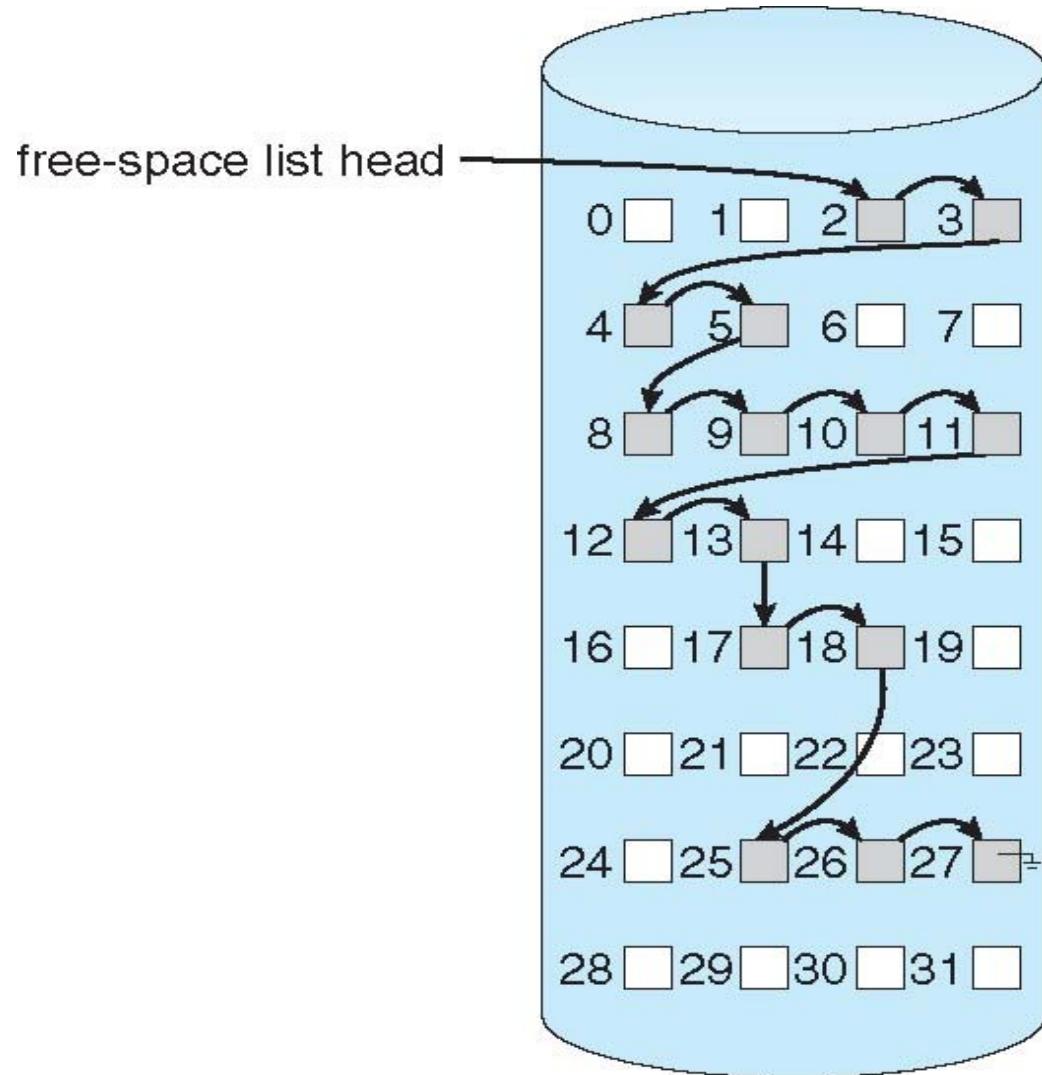
Free Space Management

- **File system maintains free-space list to track available blocks/clusters**
 - Bit vector or bit map (n blocks)
 - Or Linked list

Free Space Management: bit vector

- Each block is represented by 1 bit.
- If the block is free, the bit is 1; if the block is allocated, the bit is 0.
 - For example, consider a disk where blocks 2, 3, 4, 5, 8, 9, 10, 11, 12, 13, 17 18, 25, 26, and 27 are free and the rest of the blocks are allocated. The free-space bitmap would be
0011100111110001100000011100000 ...
- A 1- TB disk with 4- KB blocks would require 32 MB ($2^{40} / 2^{12} = 2^{28}$ bits = 2^{25} bytes = 2^5 MB) to store its bitmap

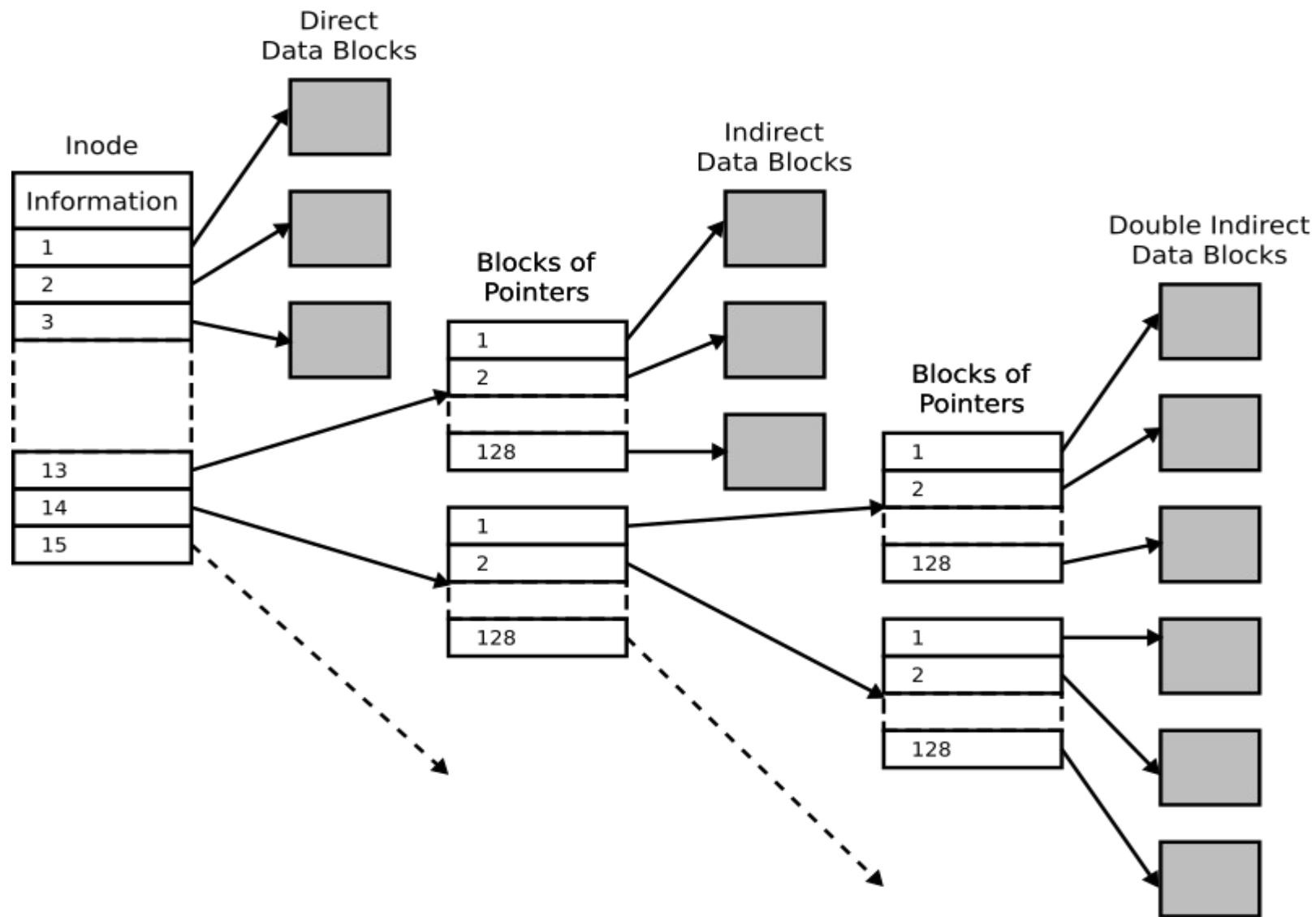
Free Space Management: Linked list (not in memory, on disk!)



Ext2 FS layout

```
struct ext2_inode {  
    __le16 i_mode;    /* File mode */  
    __le16 i_uid;     /* Low 16 bits of Owner Uid */  
    __le32 i_size;    /* Size in bytes */  
    __le32 i_atime;   /* Access time */  
    __le32 i_ctime;   /* Creation time */  
    __le32 i_mtime;   /* Modification time */  
    __le32 i_dtime;   /* Deletion Time */  
    __le16 i_gid;     /* Low 16 bits of Group Id */  
    __le16 i_links_count; /* Links count */  
    __le32 i_blocks;  /* Blocks count */  
    __le32 i_flags;   /* File flags */
```

Inode in ext2



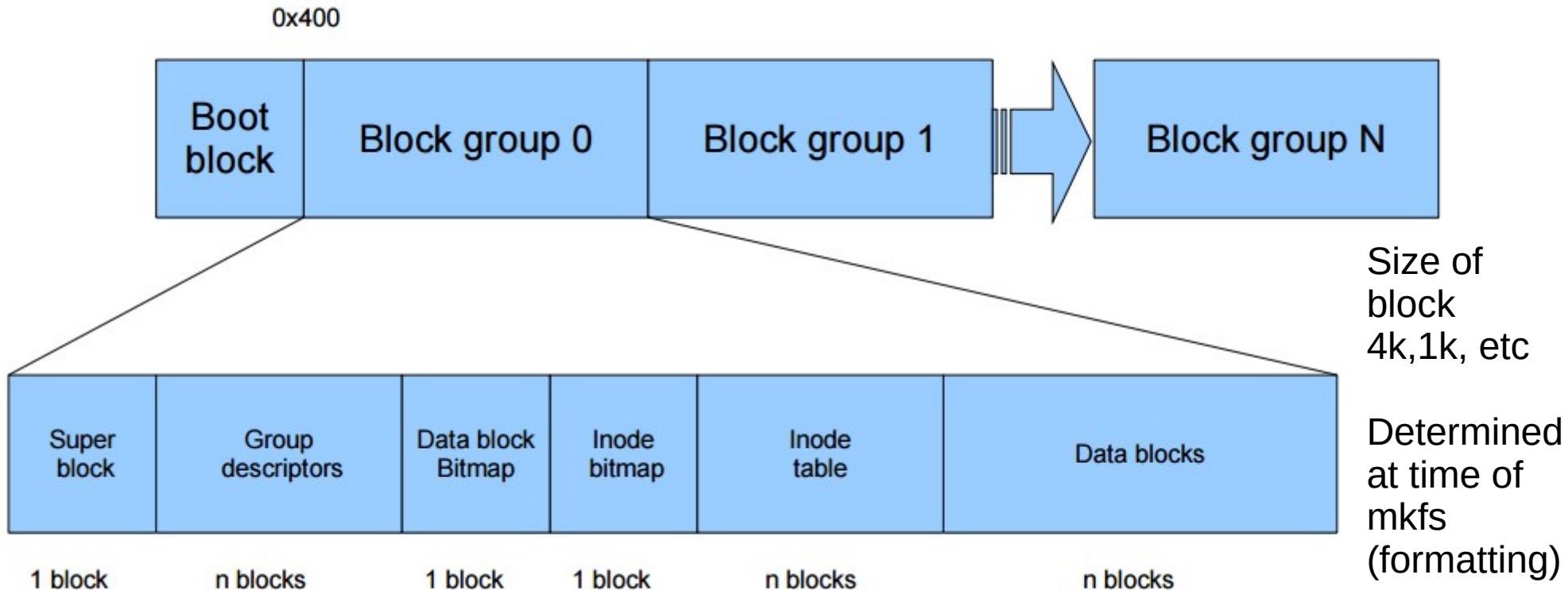
```
struct ext2_inode {  
    ...  
    union {  
        struct {  
            __le32 l_i_reserved1;  
        } linux1;  
        struct {  
            __le32 h_i_translator;  
        } hurd1;  
        struct {  
            __le32 m_i_reserved1;  
        } masix1;  
    } osd1;          /* OS dependent 1 */  
    __le32 i_block[EXT2_N_BLOCKS];/* Pointers to blocks */  
    __le32 i_generation; /* File version (for NFS) */  
    __le32 i_file_acl; /* File ACL */  
    __le32 i_dir_acl; /* Directory ACL */  
    __le32 i_faddr;   /* Fragment address */
```

```
struct ext2_inode {  
    ...  
    union {  
        struct {  
            __u8  l_i_frag; /* Fragment number */      __u8  l_i_fsize; /* Fragment size */  
            __u16 i_pad1;           __le16 l_i_uid_high; /* these 2 fields */  
            __le16 l_i_gid_high; /* were reserved2[0] */  
            __u32 l_i_reserved2;  
        } linux2;  
        struct {  
            __u8  h_i_frag; /* Fragment number */      __u8  h_i_fsize; /* Fragment size */  
            __le16 h_i_mode_high;           __le16 h_i_uid_high;  
            __le16 h_i_gid_high;  
            __le32 h_i_author;  
        } hurd2;  
        struct {  
            __u8  m_i_frag; /* Fragment number */      __u8  m_i_fsize; /* Fragment size */  
            __u16 m_pad1;           __u32  m_i_reserved2[2];  
        } masix2;  
    } osd2;          /* OS dependent 2 */
```

Ext2 FS Layout: Entries in directory's data blocks

	inode	rec_len	file_type	name_len	name						
0	21	12	1	2	.	\0	\0	\0			
12	22	12	2	2	.	.	\0	\0			
24	53	16	5	2	h	o	m	e	1	\0	\0
40	67	28	3	2	u	s	r	\0			
52	0	16	7	1	o	l	d	f	i	l	e
68	34	12	4	2	s	b	i	n			

Ext2 FS Layout



Calculations done by “mkfs” like this

- **Block size = 4KB (specified to mkfs)**
- **Number of total blocks = size of partition / 4KB**
 - How to get size of partition ?
- **$4KB = 4 * 1024 * 8 = 32768$ bits**
- **Data Block Bitmap, Inode Bitmap are always one block**
- **So**
 - size of a group is 32,768 Blocks
 - #groups = #blocks-in-partition / 32,768

```
struct ext2_super_block {
    __le32 s_inodes_count; /* Inodes count */
    __le32 s_blocks_count; /* Blocks count */
    __le32 s_r_blocks_count; /* Reserved blocks count */
    __le32 s_free_blocks_count; /* Free blocks count */
    __le32 s_free_inodes_count; /* Free inodes count */
    __le32 s_first_data_block; /* First Data Block */
    __le32 s_log_block_size; /* Block size */
    __le32 s_log_frag_size; /* Fragment size */
    __le32 s_blocks_per_group; /* # Blocks per group */
    __le32 s_frags_per_group; /* # Fragments per group */
    __le32 s_inodes_per_group; /* # Inodes per group */
    __le32 s_mtime; /* Mount time */
    __le32 s_wtime; /* Write time */
    __le16 s_mnt_count; /* Mount count */
    __le16 s_max_mnt_count; /* Maximal mount count */
    __le16 s_magic; /* Magic signature */
    __le16 s_state; /* File system state */
    __le16 s_errors; /* Behaviour when detecting errors */
```

```
struct ext2_super_block {  
...  
    __le16 s_minor_rev_level; /* minor revision level */  
    __le32 s_lastcheck;      /* time of last check */  
    __le32 s_checkinterval; /* max. time between checks */  
    __le32 s_creator_os;    /* OS */  
    __le32 s_rev_level;     /* Revision level */  
    __le16 s_def_resuid;   /* Default uid for reserved blocks */  
    __le16 s_def_resgid;   /* Default gid for reserved blocks */  
    __le32 s_first_ino;    /* First non-reserved inode */  
    __le16 s_inode_size;   /* size of inode structure */  
    __le16 s_block_group_nr; /* block group # of this superblock */  
    __le32 s_feature_compat; /* compatible feature set */  
    __le32 s_feature_incompat; /* incompatible feature set */  
    __le32 s_feature_ro_compat; /* readonly-compatible feature set */  
    __u8 s_uuid[16]; /* 128-bit uuid for volume */  
    char s_volume_name[16]; /* volume name */  
    char s_last_mounted[64]; /* directory where last mounted */  
    __le32 s_algorithm_usage_bitmap; /* For compression */
```

```
struct ext2_super_block {  
...  
    __u8    s_prealloc_blocks; /* Nr of blocks to try to preallocate*/  
    __u8    s_prealloc_dir_blocks; /* Nr to preallocate for dirs */  
    __u16   s_padding1;  
/*  
 * Journaling support valid if EXT3_FEATURE_COMPAT_HAS_JOURNAL set.  
 */  
    __u8    s_journal_uuid[16]; /* uuid of journal superblock */  
    __u32   s_journal_inum;    /* inode number of journal file */  
    __u32   s_journal_dev;    /* device number of journal file */  
    __u32   s_last_orphan;    /* start of list of inodes to delete */  
    __u32   s_hash_seed[4];    /* HTREE hash seed */  
    __u8    s_def_hash_version; /* Default hash version to use */  
    __u8    s_reserved_char_pad;  
    __u16   s_reserved_word_pad;  
    __le32  s_default_mount_opts;  
    __le32  s_first_meta_bg;   /* First metablock block group */  
    __u32   s_reserved[190];   /* Padding to the end of the block */
```

```
struct ext2_group_desc
{
    __le32 bg_block_bitmap;      /* Blocks bitmap block */
    __le32 bg_inode_bitmap;     /* Inodes bitmap block */
    __le32 bg_inode_table;     /* Inodes table block */
    __le16 bg_free_blocks_count; /* Free blocks count */
    __le16 bg_free_inodes_count; /* Free inodes count */
    __le16 bg_used_dirs_count; /* Directories count */
    __le16 bg_pad;
    __le32 bg_reserved[3];
};
```

Traversal / path-name resolution

```
//resolving /a/b

s = read_superblock(); // struct
g = read_bg_descriptors(); // array
inode getinode(int n) {
    calculate the block number for n'th inode
    (using info from superblock, bg descriptors, block-size etc)
    read that block
    extract inode from block
    return inode
}
ino = 2
i = getinode(ino) ; //root
while (path not complete) {
    if (i is directory and path not complete)
        x = get-pathname-component(path); // give "a" from "/a/b", then "b" , etc
        read-data blocks of i'th inode
        search for x in the data-blocks
    if found
        ino = inode for found entry
    else
        return not-found
    else
        return not-found
}
```

Let's see a program to read superblock of an ext2
file system.

Synchronization

My formulation

- OS = data structures + synchronization
- Synchronization problems make writing OS code challenging
- Demand exceptional coding skills

Race problem

```
long c = 0, c1 = 0, c2 = 0, run = 1;  
void *thread1(void *arg) {  
    while(run == 1) {  
        c++;  
        c1++;  
    }  
}  
void *thread2(void *arg) {  
    while(run == 1) {  
        c++;  
        c2++;  
    }  
}
```

```
int main() {  
    pthread_t th1, th2;  
    pthread_create(&th1, NULL, thread1,  
NULL);  
    pthread_create(&th2, NULL, thread2,  
NULL);  
    //fprintf(stdout, "Ending main\n");  
    sleep(2);  
    run = 0;  
    fprintf(stdout, "c = %ld c1+c2 = %ld  
c1 = %ld c2 = %ld \n", c, c1+c2, c1, c2);  
    fflush(stdout);  
}
```

Race problem

- On earlier slide
 - Value of c should be equal to $c_1 + c_2$, but it is not!
 - Why?
- There is a “race” between thread1 and thread2 for updating the variable c
- thread1 and thread2 may get scheduled in any order and *interrupted* any point in time
- The changes to c are not atomic!
 - What does that mean?

Race problem

- C++, when converted to assembly code, could be

```
mov c, r1  
add r1, 1  
mov r1, c
```

- Now following sequence of instructions is possible among thread1 and thread2

```
thread1: mov c, r1  
thread2: mov c, r1  
thread1: add r1, 1  
thread1: mov r1, c  
thread2: add r1, 1  
thread2: mov r1, c
```

- What will be value in c, if initially c was, say 5?

- It will be 6, when it is expected to be 7. Other variations also possible.

Races: reasons

- **Interruptible kernel**
 - If entry to kernel code does not disable interrupts, then modifications to any kernel data structure can be left incomplete
 - This introduces concurrency
- **Multiprocessor systems**
 - On SMP systems: memory is shared, kernel and process code run on all processors
 - Same variable can be updated parallelly (not concurrently)
- **What about non-interruptible kernel on multiprocessor systems?**
- **What about non-interruptible kernel on uniprocessor systems?**

Critical Section problem

```
do {  
    entry section  
    critical section  
    exit section  
    remainder section  
} while (TRUE);
```

Figure 6.1 General structure of a typical process P .

Critical Section Problem

- Consider system of n processes {p₀, p₁, ... p_{n-1}}
- Each process has critical section segment of code
 - Process may be changing common variables, updating table, writing file, etc
 - When one process in critical section, no other may be in its critical section
- Critical section problem is to design protocol to solve this
- Each process must ask permission to enter critical section in entry section, may follow critical section with exit section, then remainder section
- Especially challenging with preemptive kernels

Expected solution characteristics

- **1. Mutual Exclusion**
 - If process P_i is executing in its critical section, then no other processes can be executing in their critical sections
- **2. Progress**
 - If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely
- **3. Bounded Waiting**
 - A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted
 - Assume that each process executes at a nonzero speed
 - No assumption concerning relative speed of the n processes

suggested solution - 1

```
int flag = 1;  
void *thread1(void *arg) {  
    while(run == 1) {  
        while(flag == 0)  
            ;  
        flag = 0;  
        c++;  
        flag = 1;  
        c1++;  
    }  
}
```

- **What's wrong here?**
- **Assumes that**
**while(flag ==) ; flag
= 0**
will be atomic

suggested solution - 2

```
int flag = 0;  
  
void *thread1(void *arg) {  
    while(run == 1) {  
        if(flag)  
            c++;  
        else  
            continue;  
        c1++;  
        flag = 0;  
    }  
}  
  
void *thread2(void *arg) {  
    while(run == 1) {  
        if(!flag)  
            c++;  
        else  
            continue;  
        c2++;  
        flag = 1;  
    }  
}
```

Peterson's solution

- Two process solution
- Assume that the LOAD and STORE instructions are atomic; that is, cannot be interrupted
- The two processes share two variables:
 - int turn;
 - Boolean flag[2]
- The variable turn indicates whose turn it is to enter the critical section
- The flag array is used to indicate if a process is ready to enter the critical section. flag[i] = true implies that process Pi is ready!

Peterson's solution

```
do {
    flag[i] = TRUE;
    turn = j;
    while (flag[j] && turn == j)
        ;
    critical section
    flag[i] = FALSE;
    remainder section
} while (TRUE);
```

- Provable that
 - Mutual exclusion is preserved
 - Progress requirement is satisfied
 - Bounded-waiting requirement is met

Hardware solution – the one actually implemented

- Many systems provide hardware support for critical section code
- Uniprocessors – could disable interrupts
 - Currently running code would execute without preemption
 - Generally too inefficient on multiprocessor systems
 - Operating systems using this not broadly scalable
- Modern machines provide special atomic hardware instructions
 - Atomic = non-interruptable
 - Either test memory word and set value
 - Or swap contents of two memory words
 - Basically two operations (read/write) done atomically in hardware

Solution using test-and-set

```
lock = false; //global  
  
do {  
    while ( TestAndSet (&lock )  
           ; // do nothing  
    //  critical section  
    lock = FALSE;  
    //  remainder section  
} while (TRUE);
```

Definition:

```
boolean TestAndSet (boolean  
                    *target)  
{  
    boolean rv = *target;  
    *target = TRUE;  
    return rv;  
}
```

Solution using swap

```
lock = false; //global  
  
do {  
    key = true  
    while ( key == true))  
        swap(&lock, &key)  
        // critical section  
    lock = FALSE;  
    // remainder section  
} while (TRUE);
```

Spinlock

- A lock implemented to do ‘busy-wait’
- Using instructions like T&S or Swap
- As shown on earlier slides

```
spinlock(int *lock){  
    While(test-and-set(lock))  
        ;  
}  
  
spinunlock(lock *lock) {  
    *lock = false;  
}
```

Bounded wait M.E. with T&S

```
do {
    waiting[i] = TRUE;
    key = TRUE;
    while (waiting[i] && key)
        key = TestAndSet(&lock);
    waiting[i] = FALSE;
    // critical section
    j = (i + 1) % n;
    while ((j != i) && !waiting[j])
        j = (j + 1) % n;
    if (j == i)
        lock = FALSE;
    else
        waiting[j] = FALSE;
    // remainder section
} while (TRUE);
```

sleep-locks

- **Spin locks result in busy-wait**
- **CPU cycles wasted by waiting processes/threads**
- **Solution – threads keep waiting for the lock to be available**
 - Move thread to wait queue
 - The thread holding the lock will wake up one of them

Sleep locks/mutexes

```
//ignore syntactical issues  
typedef struct mutex {  
    int islocked;  
    int spinlock;  
    waitqueue q;  
}mutex;  
wait(mutex *m) {  
    spinlock(m->spinlock);  
    while(m->islocked)  
        Block(m, m->spinlock)  
    lk->islocked = 1;  
    spinunlock(m->spinlock);  
}
```

```
Block(mutex *m, spinlock *sl) {  
    currprocess->state = WAITING  
    move current process to m->q  
    spinunlock(sl);  
    Sched();  
    spinlock(sl);  
}  
release(mutex *m) {  
    spinlock(m->spinlock);  
    m->islocked = 0;  
    Some process in m->queue  
    =RUNNABLE;  
    spinunlock(m->spinlock);  
}
```

Some thumb-rules of spinlocks

- **Never block a process holding a spinlock !**

- **Typical code:**

```
while(condition)
    { Spin-unlock()
      Schedule()
      Spin-lock()
    }
```

- **Hold a spin lock for only a short duration of time**

- **Spinlocks are preferable on multiprocessor systems**
- **Cost of context switch is a concern in case of sleep-wait locks**
- **Short = < 2 context switches**

Locks in xv6 code

struct spinlock

// Mutual exclusion lock.

```
struct spinlock {
```

```
    uint locked;      // Is the lock held?
```

// For debugging:

```
    char *name;      // Name of lock.
```

```
    struct cpu *cpu; // The cpu holding the lock.
```

```
    uint pcs[10];    // The call stack (an array of program counters)
```

```
                           // that locked the lock.
```

```
};
```

spinlocks in xv6 code

```
struct {  
    struct spinlock lock;  
    struct buf buf[NBUF];  
    struct buf head;  
} bcache;  
struct {  
    struct spinlock lock;  
    struct file file[NFILE];  
} ftable;  
struct {  
    struct spinlock lock;  
    struct inode inode[NINODE];  
} icache;  
struct sleeplock {  
    uint locked;      // Is the lock held?  
    struct spinlock sl;
```

```
static struct spinlock idelock;  
struct {  
    struct spinlock lock;  
    int use_lock;  
    struct run *freelist;  
} kmem;  
struct log {  
    struct spinlock lock;  
...}  
struct pipe {  
    struct spinlock lock;  
...}  
struct {  
    struct spinlock lock;  
    struct proc proc[NPROC];  
} ptable;  
struct spinlock tickslock;
```

```

static inline uint
xchg(volatile uint *addr, uint newval)
{
    uint result;
    // The + in "+m" denotes a read-modify-
    // write operand.
    asm volatile("lock; xchgl %0, %1" :
        "+m" (*addr), "=a" (result) :
        "1" (newval) :
        "cc");
    return result;
}

struct spinlock {
    uint locked;      // Is the lock held?

    // For debugging:
    char *name;      // Name of lock.
    struct cpu *cpu; // The cpu holding the
                     // lock.

    uint pcs[10];    // The call stack (an array
                     // of program counters) that locked the lock.
};

```

Spinlock in xv6

```

void acquire(struct spinlock *lk)
{
    pushcli(); // disable interrupts to
               // avoid deadlock.

    // The xchg is atomic.
    while(xchg(&lk->locked, 1) != 0)
        ;

    //extra debugging code
}

void release(struct spinlock *lk)
{
    //extra debugging code
    asm volatile("movl $0, %0" :
        "+m" (lk->locked) : );
    popcli();
}

```

```

Void acquire(struct spinlock *lk)
{
    pushcli(); // disable interrupts to avoid deadlock.
    if(holding(lk))
        panic("acquire");
    .....
void pushcli(void)
{
    int eflags;
    eflags = readeflags();
    cli();
    if(mycpu()->ncli == 0)
        mycpu()->intena = eflags & FL_IF;
    mycpu()->ncli += 1;
}
static inline uint
readeflags(void)
{
    uint eflags;
    asm volatile("pushfl; popl %0" : "=r" (eflags));
    return eflags;
}

```

spinlocks

- **Pushcli() - disable interrupts on that processor**
- **One after another many acquire() can be called on different spinlocks**
 - Keep a count of them in mycpu()->ncli

```

void
release(struct spinlock *lk)
{
...
    asm volatile("movl $0, %0" : "+m" (lk->locked) : );
    popcli();
}

.

Void popcli(void)
{
    if(readeflags()&FL_IF)
        panic("popcli - interruptible");
    if(--mycpu()->ncli < 0)
        panic("popcli");
    if(mycpu()->ncli == 0 && mycpu()->intena)
        sti();
}

```

spinlocks

- **Popcli()**

- **Restore interrupts if last popcli() call restores ncli to 0 & interrupts were enabled before pushcli() was called**

spinlocks

- Always disable interrupts while acquiring spinlock
 - Suppose **iderw** held the **idelock** and then got interrupted to run **ideintr**.
 - **Ideintr** would try to lock **idelock**, see it was held, and wait for it to be released.
 - In this situation, **idelock** will never be released
 - Deadlock
- General OS rule: if a spin-lock is used by an interrupt handler, a processor must never hold that lock with interrupts enabled
- Xv6 rule: when a processor enters a spin-lock critical section, xv6 always ensures interrupts are disabled on that processor.

sleeplocks

- **Sleeplocks don't spin. They move a process to a wait-queue if the lock can't be acquired**
- **XV6 approach to “wait-queues”**
 - Any memory address serves as a “wait channel”
 - The sleep() and wakeup() functions just use that address as a ‘condition’
 - There are no per condition process queues! Just one global queue of processes used for scheduling, sleep, wakeup etc. --> Linear search everytime !
 - **costly, but simple**

```

void
sleep(void *chan, struct spinlock *lk)
{
    struct proc *p = myproc();
    ....
    if(lk != &ptable.lock){
        acquire(&ptable.lock);
        release(lk);
    }
    p->chan = chan;
    p->state = SLEEPING;
    sched();
    // Reacquire original lock.
    if(lk != &ptable.lock){
        release(&ptable.lock);
        acquire(lk);
    }
}

```

sleep()

- At call must hold lock on the resource on which you are going to sleep
- since you are going to change p-> values & call sched(), hold ptable.lock if not held
- p->chan = given address remembers on which condition the process is waiting
- call to sched() blocks the process

Calls to sleep() : examples of “chan” (output from cscope)

0 console.c

consoleread 251

sleep(&input.r, &cons.lock);

2 ide.c iderw

169 sleep(b, &idelock);

3 log.c begin_op

131 sleep(&log, &log.lock);

6 pipe.c piperead

111 sleep(&p->nread, &p->lock);

7 proc.c wait

317 sleep(curproc,
&phtable.lock);

8 sleeplock.c

acquiresleep 28

sleep(lk, &lk->lk);

9 sysproc.c

sys_sleep 74

sleep(&ticks, &tickslock);

```

void wakeup(void *chan)
{
    acquire(&ptable.lock);
    wakeup1(chan);
    release(&ptable.lock);
}

static void wakeup1(void *chan)
{
    struct proc *p;

    for(p = ptable.proc; p <
&ptable.proc[NPROC]; p++)
        if(p->state == SLEEPING &&
p->chan == chan)
            p->state = RUNNABLE;
}

```

Wakeup()

- Acquire ptable.lock since you are going to change ptable and p-> values
- just linear search in process table for a process where p->chan is given address
- Make it runnable

sleeplock

// Long-term locks for processes

struct sleeplock {

 uint locked; // Is the lock held?

 struct spinlock sl; // spinlock protecting this sleep lock

// For debugging:

 char *name; // Name of lock.

 int pid; // Process holding lock

};

Sleeplock acquire and release

```
void
acquiresleep(struct sleeplock *lk)
{
    acquire(&lk->lk);
    while (lk->locked) {
        /* Abhijit: interrupts are not disabled in
         sleep !*/
        sleep(lk, &lk->lk);
    }
    lk->locked = 1;
    lk->pid = myproc()->pid;
    release(&lk->lk);
}
```

```
void
releasesleep(struct sleeplock
*lk)
{
    acquire(&lk->lk);
    lk->locked = 0;
    lk->pid = 0;
    wakeup(lk);
    release(&lk->lk);
}
```

Where are sleeplocks used?

- **struct buf**
 - waiting for I/O on this buffer
- **struct inode**
 - waiting for I/o to this inode
- Just two !

Sleeplocks issues

- sleep-locks support yielding the processor during their critical sections.
- This property poses a design challenge:
 - if thread T1 holds lock L1 and has yielded the processor (waiting for some other condition),
 - and thread T2 wishes to acquire L1,
 - we have to ensure that T1 can execute
 - while T2 is waiting so that T1 can release L1.
 - T2 can't use the spin-lock acquire function here: it spins with interrupts turned off, and that would prevent T1 from running.
- To avoid this deadlock, the sleep-lock acquire routine (called `acquiresleep`) yields the processor while waiting, and does not disable interrupts.

Sleep-locks leave interrupts enabled, they cannot be used in interrupt handlers.

More needs of synchronization

- Not only critical section problems
- Run processes in a particular order
- Allow multiple processes read access, but only one process write access
- Etc.

Semaphore

- **Synchronization tool that does not require busy waiting**
 - **Semaphore S – integer variable**
 - **Two standard operations modify S: wait() and signal()**
 - Originally called P() and V()
 - **Less complicated**
- Can only be accessed via two indivisible (atomic) operations
- ```
wait (S) {
 while S <= 0
 ; // no-op
 S--;
}
signal (S) {
 S++;
}
--> Note this is Signal() on a semaphore, different froms signal system call
```

# Semaphore for synchronization

- **Counting** semaphore – integer value can range over an unrestricted domain
- **Binary** semaphore – integer value can range only between 0 and 1; can be simpler to implement

Also known as **mutex locks**

- Can implement a counting semaphore **S** as a binary semaphore
- Provides mutual exclusion

```
Semaphore mutex; // initialized to 1
do {
 wait (mutex);
 // Critical Section
 signal (mutex);
 // remainder section
} while (TRUE)
```

# **Different uses of semaphores**

# For mutual exclusion

**/\*During initialization\*/**

**semaphore sem;**

**initsem (&sem, 1);**

**/\* On each use\*/**

**P (&sem);**

**Use resource;**

**V (&sem);**

# Event-wait

```
/* During initialization */
semaphore event;
initsem (&event, 0); /* probably at boot time */

/* Code executed by thread that must wait on event */
P (&event); /* Blocks if event has not occurred */
/* Event has occurred */
V (&event); /* So that another thread may wake up */
/* Continue processing */

/* Code executed by another thread when event occurs */
V (&event); /* Wake up one thread */
```

# Control countable resources

**/\* During initialization \*/**

semaphore counter;

initsem (&counter, resourceCount);

**/\* Code executed to use the resource \*/**

P (&counter); /\* Blocks until resource is available \*/

Use resource; /\* Guaranteed to be available now \*/

V (&counter); /\* Release the resource \*/

# Semaphore implementation

```
Wait(sem *s) {
 while(s <=0)
 block(); // could be ";"
 s--;
}

signal(sem *s) {
 s++;
}
```

- Left side – expected behaviour
- Both the wait and signal should be atomic.
- This is the semantics of the semaphore.

# Semaphore implementation? - 1

```
struct semaphore {
 int val;
 spinlock sl;
};
sem_init(semaphore *s, int initval) {
 s->val = initval;
 s->sl = 0;
}
wait(semaphore *s) {
 spinlock(&(s->sl));
 while(s->val <=0)
 ;
 (s->val)--;
 spinunlock(&(s->sl));
}
```

```
signal(semaphore *s) {
 spinlock(*(s->sl));
 (s->val)++;
 spinunlock(*(s->sl));
}
- suppose 2 processes trying wait.
val = 1;
Th1: spinlock Th2: spinlock-waits
Th1: while -> false, val-- => 0; spinunlock;
Th2: spinlock success; while() -> true, loops;
Th1: is done with critical section, it calls signal. it calls spinlock() -> wait.
Who is holding spinlock-> Th2. It is waiting for val > 0. Who can set value > 0 , ans: Th1, and Th1 is waiting for spinlock which is held by Th2.
circular wait. Deadlock.
None of them will proceed.
```

# Semaphore implementation? - 2

```
struct semaphore {
 int val;
 spinlock sl;
};
sem_init(semaphore *s, int initval) {
 s->val = initval;
 s->sl = 0;
}
signal(semaphore *s) {
 spinlock(*(s->sl));
 (s->val)++;
 spinunlock(*(s->sl));
}
```

```
wait(semaphore *s) {
 spinlock(&(s->sl));
 while(s->val <=0) {
 spinunlock(&(s->sl));
 spinlock(&(s->sl));
 }
 (s->val)--;
 spinunlock(&(s->sl));
}
```

**Problem:** race in spinlock of while loop and signal's spinlock.  
**Bounded wait not guaranteed.**  
**Spinlocks are not good for a long wait.**

# Semaphore implementation? - 3, idea

```
struct semaphore {
 int val;
 spinlock sl;
};
sem_init(semaphore *s, int initval) {
 s->val = initval;
 s->sl = 0;
}
block() {
 put this current process on wait-q;
 schedule();
}

wait(semaphore *s) {
 spinlock(&(s->sl));
 while(s->val <=0) {
 Block();
 }
 (s->val)--;
 spinunlock(&(s->sl));
}

signal(seamphore *s) {
 spinlock(*(s->sl));
 (s->val)++;
 spinunlock(*(s->sl));
}
```

# Semaphore implementation? - 3a

```
struct semaphore {
 int val;
 spinlock sl;
 list l;
};
sem_init(semaphore *s, int initval) {
 s->val = initval;
 s->sl = 0;
}
block(semaphore *s) {
 listappend(s->l, current);
 schedule();
}
problem is that block() will be called
without holding the spinlock and the
access to the list is not protected.
Note that - so far we have ignored changes
to signal()
```

```
wait(semaphore *s) {
 spinlock(&(s->sl));
 while(s->val <=0) {
 spinunlock(&(s->sl));
 block(s);
 }
 (s->val)--;
 spinunlock(&(s->sl));
}
signal(seamphore *s) {
 spinlock(*(s->sl));
 (s->val)++;
 spinunlock(*(s->sl));
}
```

# Semaphore implementation? - 3b

```
struct semaphore {
 int val;
 spinlock sl;
 list l;
};
sem_init(semaphore *s, int initval) {
 s->val = initval;
 s->sl = 0;
}
block(semaphore *s) {
 listappend(s->l, current);
 spinunlock(&(s->sl));
 schedule();
}
wait(semaphore *s) {
 spinlock(&(s->sl));
 while(s->val <=0) {
 block(s);
 }
 (s->val)--;
 spinunlock(&(s->sl));
}
signal(semaphore *s) {
 spinlock(*(s->sl));
 (s->val)++;
 x = dequeue(s->sl) and enqueue(readyq, x);
 spinunlock(*(s->sl));
}
Problem: after a blocked process comes out
of the block, it does not hold the spinlock and
it's going to change the s->sl;
```

# Semaphore implementation? - 3c

```
struct semaphore {
 int val;
 spinlock sl;
 list l;
};
sem_init(semaphore *s, int initval) {
 s->val = initval;
 s->sl = 0;
}
block(semaphore *s) {
 listappend(s->l, current);
 spinunlock(&(s->sl));
 schedule();
}

wait(semaphore *s) {
 spinlock(&(s->sl)); // A
 while(s->val <=0) {
 block(s);
 spinlock(&(s->sl)); // B
 }
 (s->val)--;
 spinunlock(&(s->sl));
}
signal(semaphore *s) {
 spinlock(*(s->sl));
 (s->val)++;
 x = dequeue(s->sl) and enqueue(readyq, x);
 spinunlock(*(s->sl));
}

Question: there is race between A and B. Can we guarantee bounded wait ?
```

# Semaphore Implementation

- Must guarantee that no two processes can execute `wait()` and `signal()` on the same semaphore at the same time
- Thus, implementation becomes the critical section problem where the `wait` and `signal` code are placed in the critical section
  - Could now have busy waiting in critical section implementation
    - But implementation code is short
    - Little busy waiting if critical section rarely occupied
  - Note that applications may spend lots of time in critical sections and therefore this is not a good solution

# Semaphore in Linux

```
struct semaphore {
 raw_spinlock_t lock;
 unsigned int count;
 struct list_head wait_list;
};

static noinline void __sched
__down(struct semaphore *sem)
{
 __down_common(sem,
 TASK_UNINTERRUPTIBLE,
 MAX_SCHEDULE_TIMEOUT);
}

void down(struct semaphore *sem)
{
 unsigned long flags;

 raw_spin_lock_irqsave(&sem->lock, flags);
 if (likely(sem->count > 0))
 sem->count--;
 else
 __down(sem);
 raw_spin_unlock_irqrestore(&sem->lock,
 flags);
}
```

# Semaphore in Linux

```
static inline int __sched
__down_common(struct semaphore
*sem, long state, long timeout)
{
 struct task_struct *task = current;
 struct semaphore_waiter waiter;
 list_add_tail(&waiter.list, &sem->wait_list);
 waiter.task = task;
 waiter.up = false;
```

```
 for (;;) {
 if (signal_pending_state(state, task))
 goto interrupted;
 if (unlikely(timeout <= 0))
 goto timed_out;
 __set_task_state(task, state);
 raw_spin_unlock_irq(&sem->lock);
 timeout = schedule_timeout(timeout);
 raw_spin_lock_irq(&sem->lock);
 if (waiter.up)
 return 0;
 }
....
```

# Drawbacks of semaphores

- Need to be implemented using lower level primitives like spinlocks
- Context-switch is involved in blocking and signaling – time consuming
- Can not be used for a short critical section

# **Deadlocks**

# Deadlock

- two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes
- Let S and Q be two semaphores initialized to 1

P0

**wait (S);**

**wait (Q);**

· ·  
· ·  
· ·

**signal (S);**

**signal (Q);**

P1

**wait (Q);**

**wait (S);**

**signal (Q);**

**signal (S);**

# Example of deadlock

- Let's see the pthreads program : `deadlock.c`
- Same programme as on earlier slide, but with `pthread_mutex_lock();`

# Non-deadlock, but similar situations

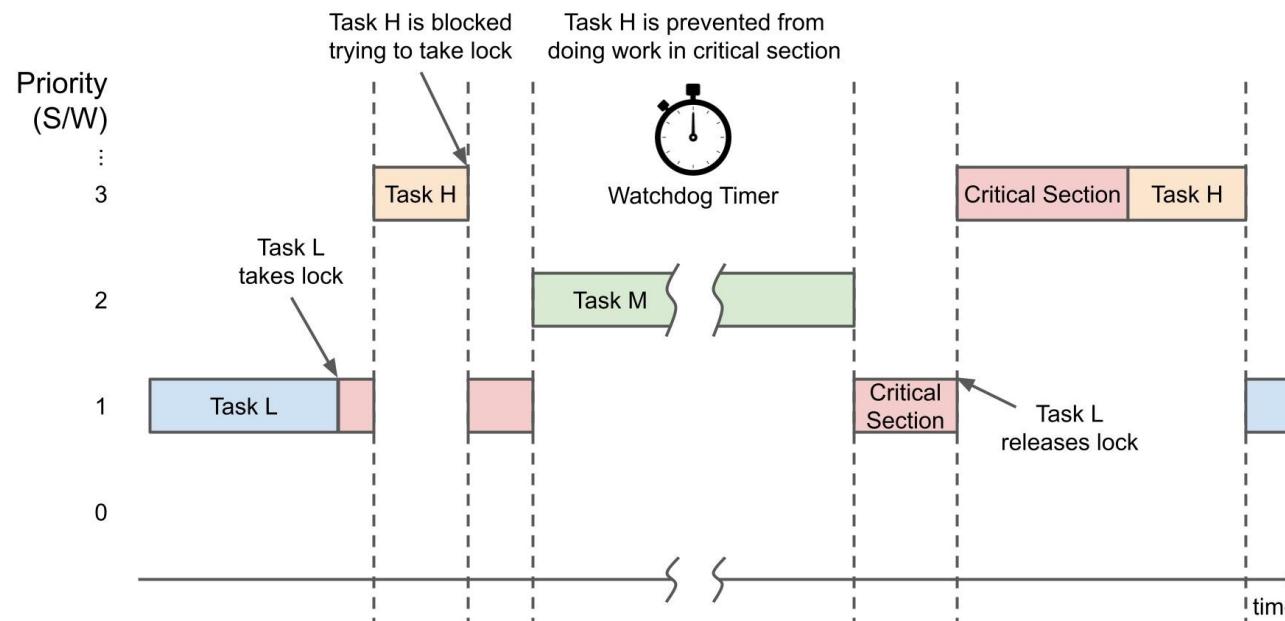
- **Starvation – indefinite blocking**
  - A process may never be removed from the semaphore queue in which it is suspended

# Non-deadlock, but similar situations

- **Priority Inversion**

- Scheduling problem when lower-priority process holds a lock needed by higher-priority process (so it can not pre-empt lower priority process), and a medium priority process (that does not need the lock) pre-empts lower priority task, denying turn to higher priority task
- Solved via priority-inheritance protocol : temporarily enhance priority of lower priority task to highest

Unbounded Priority Inversion



# Livelock

- **Similar to deadlock, but processes keep doing ‘useless work’**
- **E.g. two people meet in a corridor opposite each other**
  - Both move to left at same time
  - Then both move to right at same time
  - Keep Repeating!
- **No process able to progress, but each doing ‘some work’ (not sleeping/waiting), state keeps changing**

# Livelock example

```
#include <stdio.h>
#include <pthread.h>
struct person {
 int otherid;
 int otherHungry;
 int myid;
};
int main() {
 pthread_t th1, th2;
 struct person one, two;
 one.otherid = 2; one.myid = 1;
 two.otherid = 1; two.myid = 2;
 one.otherHungry = two.otherHungry = 1;
 pthread_create(&th1, NULL, eat, &one);
 pthread_create(&th2, NULL, eat, &two);
 printf("Main: Waiting for threads to get over\n");
 pthread_join(th1, NULL);
 pthread_join(th2, NULL);
 return 0;
}

/* thread two runs in this function */
int spoonWith = 1;
void *eat(void *param)
{
 int eaten = 0;
 struct person person= *(struct person *)param;
 while (!eaten) {
 if(spoonWith == person.myid)
 printf("%d going to eat\n", person.myid);
 else
 continue;
 if(person.otherHungry) {
 printf("You eat %d\n", person.otherid);
 spoonWith = person.otherid;
 continue;
 }
 printf("%d is eating\n", person.myid);
 break;
 }
}
```

# More on deadlocks

- **Under which conditions they can occur?**
- **How can deadlocks be avoided/prevented?**
- **How can a system recover if there is a deadlock ?**

# **System model for understanding deadlocks**

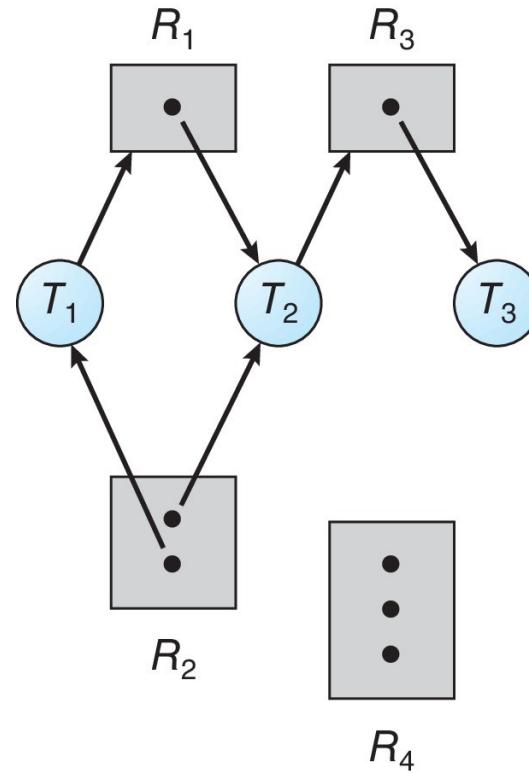
- **System consists of resources**
- **Resource types R<sub>1</sub>, R<sub>2</sub>, . . . , R<sub>m</sub>**
  - CPU cycles, memory space, I/O devices
  - Resource: Most typically a lock, synchronization primitive
- **Each resource type R<sub>i</sub> has W<sub>i</sub> instances.**
- **Each process utilizes a resource as follows:**
  - request
  - use
  - release

# Deadlock characterisation

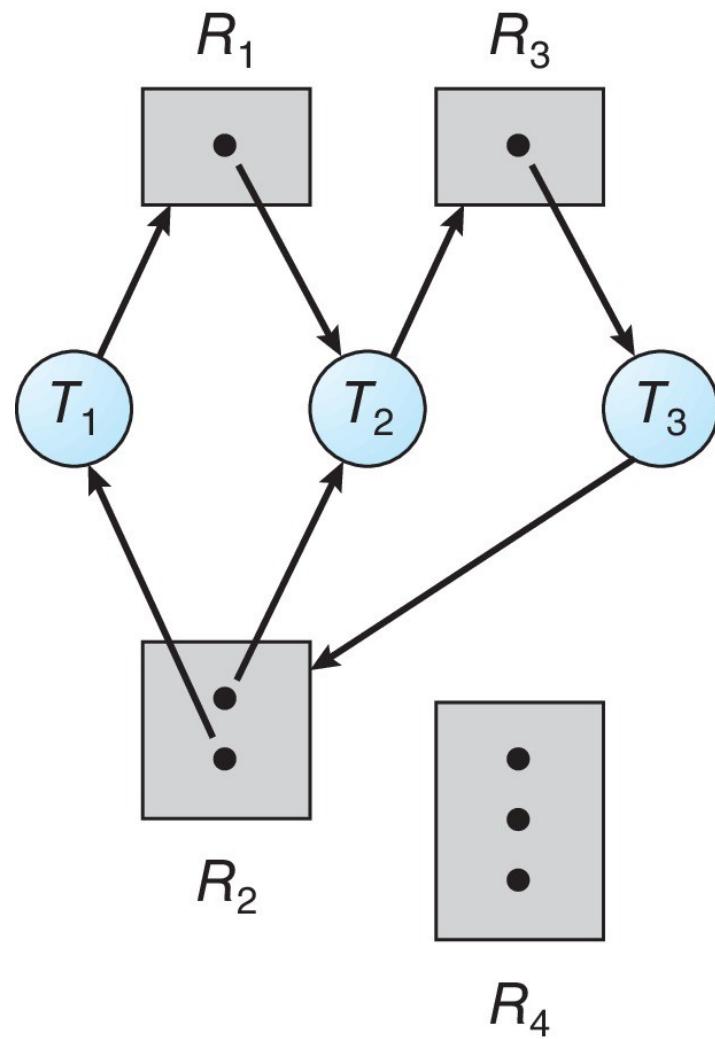
- **Deadlock is possible only if ALL of these conditions are TRUE at the same time**
  - **Mutual exclusion:** only one process at a time can use a resource
  - **Hold and wait:** a process holding at least one resource is waiting to acquire additional resources held by other processes
  - **No preemption:** a resource can be released only voluntarily by the process holding it, after that process has completed its task
  - **Circular wait:** there exists a set  $\{P_0, P_1, \dots, P_n\}$  of waiting processes such that  $P_0$  is waiting for a resource that is held by  $P_1$ ,  $P_1$  is waiting for a resource that is held by  $P_2, \dots, P_{n-1}$  is waiting for a resource that is held by  $P_n$ , and  $P_n$  is waiting for a resource that is held by  $P_0$ .

# Resource Allocation Graph Example

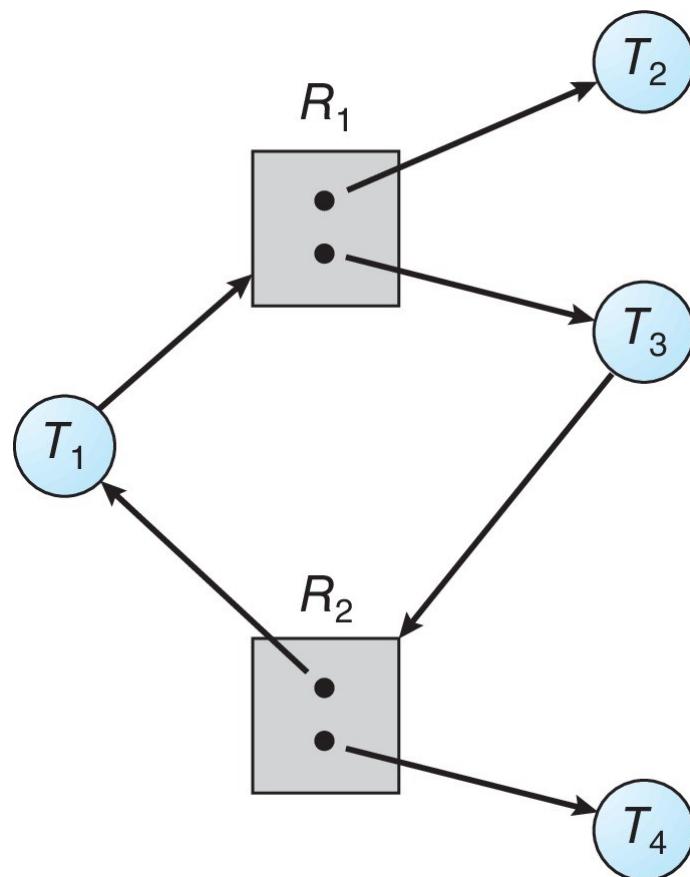
- One instance of R1
- Two instances of R2
- One instance of R3
- Three instances of R4
- T1 holds one instance of R2 and is waiting for an instance of R1
- T2 holds one instance of R1, one instance of R2, and is waiting for an instance of R3
- T3 is holding one instance of R3



# Resource Allocation Graph with a Deadlock



# Graph with a Cycle But no Deadlock



# Basic Facts

- **If graph contains no cycles -> no deadlock**
- **If graph contains a cycle :**
  - **if only one instance per resource type, then deadlock**
  - **if several instances per resource type, possibility of deadlock**

# Methods for Handling Deadlocks

- Ensure that the system will never enter a deadlock state:
  - 1) Deadlock prevention
  - 2) Deadlock avoidance
- 3) Allow the system to enter a deadlock state and then recover
- 4) Ignore the problem and pretend that deadlocks never occur in the system.

# (1) Deadlock Prevention

- **Invalidate one of the four necessary conditions for deadlock:**
- **Mutual Exclusion** – not required for sharable resources (e.g., read-only files); must hold for non-sharable resources
- **Hold and Wait** – must guarantee that whenever a process requests a resource, it does not hold any other resources
  - Require process to request and be allocated all its resources before it begins execution, or allow process to request resources only when the process has none allocated to it.
  - Low resource utilization; starvation possible

# (1) Deadlock Prevention (Cont.)

- **No Preemption:**
  - If a process that is holding some resources requests another resource that cannot be immediately allocated to it, then all resources currently being held are released
  - Preempted resources are added to the list of resources for which the process is waiting
  - Process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting
- **Circular Wait:**
  - Impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration

# (1) Deadlock prevention: Circular Wait

- **Invalidate the circular wait condition is most common.**
- **Simply assign each resource (i.e., mutex locks) a unique number.**
- **Resources must be acquired in order.**
- **If:**
  - first\_mutex is mapped to order 1**
  - second\_mutex is mapped to order 5**
  - code for thread\_two could not be written like on RHS**

```
/* thread_one runs in this function */
void *do_work_one(void *param)
{
 pthread_mutex_lock(&first_mutex);
 pthread_mutex_lock(&second_mutex);
 /**
 * Do some work
 */
 pthread_mutex_unlock(&second_mutex);
 pthread_mutex_unlock(&first_mutex);

 pthread_exit(0);
}

/* thread_two runs in this function */
void *do_work_two(void *param)
{
 pthread_mutex_lock(&second_mutex);
 pthread_mutex_lock(&first_mutex);
 /**
 * Do some work
 */
 pthread_mutex_unlock(&first_mutex);
 pthread_mutex_unlock(&second_mutex);

 pthread_exit(0);
}
```

# (1) Preventing deadlock: cyclic wait

- **Locking hierarchy : Highly preferred technique in kernels**
  - Decide an ordering among all ‘locks’
  - Ensure that on ALL code paths in the kernel, the locks are obtained in the decided order!
  - Poses coding challenges!
  - A key differentiating factor in kernels
  - Do not look at only the current lock being taken, look at all the locks the code may be holding at any given point in code!

# **(1) Prevention in Xv6: Lock Ordering**

- **lock on the directory, a lock on the new file's inode, a lock on a disk block buffer, idelock, and ptable.lock.**

## (2) Deadlock avoidance

- **Requires that the system has some additional a priori information available**
  - Processes declare resources they want, BEFORE-hand
  - Resources are always allocated by an ALLOCATOR algorithm
    - It can predict if a deadlock can happen

## (2) Deadlock avoidance

- Please see: concept of safe states, unsafe states, Banker's algorithm

# (3) Deadlock detection and recovery

- How to detect a deadlock in the system?
- The Resource-Allocation Graph is a graph. Need an algorithm to detect cycle in a graph.
- How to recover?
  - Abort all processes or abort one by one?
  - Which processes to abort?
    - Priority ?
    - Time spent since forked()?
    - Resources used?
    - Resources needed?
    - Interactive or not?
    - How many need to be terminated?

# **“Condition” Synchronization Tool**

# What is condition variable?

- A variable with a sleep queue
- Threads can sleep on it, and wake-up all remaining

Struct condition {

    Proc \*next

    Proc \*prev

    Spinlock \*lock

}

Different variables of this type can be used as different  
'conditions'

# Code for condition variables

```
//Spinlock s is held before calling wait
void wait (condition *c, spinlock_t *s)
{
 spin_lock (&c->listLock);
 add self to the linked list;
 spin_unlock (&c->listLock);
 spin_unlock (s); /* release
spinlock before blocking */
 swtch(); /* perform context switch */
 /* When we return from swtch, the
event has occurred */
 spin_lock (s); /* acquire the spin
lock again */
 return;
}
```

```
void do_signal (condition *c)
/*Wakeup one thread waiting on the condition*/
{
 spin_lock (&c->listLock);
 remove one thread from linked list, if it is nonempty;
 spin_unlock (&c->listLock);
 if a thread was removed from the list, make it
 runnable;
 return;
}
void do broadcast (condition *c)
/*Wakeup al lthreads waiting on the condition*/
{
 spin_lock (&c->listLock);
 while (linked list is nonempty) {
 remove a thread from linked list;
 make it runnable;
 }
 spin_unlock (&c->listLock);
}
```

# Semaphore implementation using condition variables?

- Is this possible?
- Can we try it?

```
typedef struct semaphore {
 //something
 condition c;
}semaphore;
```

- Now write code for semaphore P() and V()

# **Classical Synchronization Problems**

# Bounded-Buffer Problem

- **Producer and consumer processes**
  - N buffers, each can hold one item
- **Producer produces ‘items’ to be consumed by consumer , in the bounded buffer**
- **Consumer should wait if there are no items**
- **Producer should wait if the ‘bounded buffer’ is full**

# Bounded-Buffer Problem: solution with semaphores

- **Semaphore mutex initialized to the value 1**
- **Semaphore full initialized to the value 0**
- **Semaphore empty initialized to the value N**

# Bounded-buffer problem

The structure of the producer process

```
do {
 // produce an item in nextp
 wait (empty);
 wait (mutex);
 // add the item to the buffer
 signal (mutex);
 signal (full);
} while (TRUE);
```

The structure of the Consumer process

```
do {
 wait (full);
 wait (mutex);
 // remove an item from
 // buffer to nextc
 signal (mutex);
 signal (empty);
 // consume item in nextc
} while (TRUE);
```

# Bounded buffer problem

- Example : pipe()
- Let's see code of pipe in xv6 – a solution using sleeplocks

# Readers-Writers problem

- **A data set is shared among a number of concurrent processes**
  - Readers – only read the data set; they do not perform any updates
  - Writers – can both read and write
- **Problem – allow multiple readers to read at the same time**
  - Only one single writer can access the shared data at the same time
- **Several variations of how readers and writers are treated – all involve priorities**
- **Shared Data**
  - Data set
  - Semaphore mutex initialized to 1
  - Semaphore wrt initialized to 1
  - Integer readcount initialized to 0

## The structure of a writer process

```
do {
 wait (wrt) ;
 // writing is performed
 signal (wrt) ;
} while (TRUE);
```

## The structure of a reader process

```
do {
 wait (mutex) ;
 readcount ++ ;
 if (readcount == 1)
 wait (wrt) ;
 signal (mutex)
 // reading is performed
 wait (mutex) ;
 readcount -- ;
 if (readcount == 0)
 signal (wrt) ;
 signal (mutex) ;
} while (TRUE);
```

# Readers-Writers problem

# **Readers-Writers Problem Variations**

- **First variation – no reader kept waiting unless writer has permission to use shared object**
- **Second variation – once writer is ready, it performs write asap**
- **Both may have starvation leading to even more variations**
- **Problem is solved on some systems by kernel providing reader-writer locks**

# Reader-write lock

- A lock with following operations on it
  - Lockshared()
  - Unlockshared()
  - LockExcl()
  - UnlockExcl()
- Possible additions
  - Downgrade() -> from excl to shared
  - Upgrade() -> from shared to excl

# Code for reader-writer locks

```
struct rwlock {
 int nActive; /* num of active
readers, or -1 if a writer is
active */

 int nPendingReads;
 int nPendingWrites;
 spinlock_t sl;
 condition canRead;
 condition canWrite;
};
```

```
void lockShared (struct rwlock *r)
{
 spin_lock (&r->sl);
 r->nPendingReads++;
 if (r->nPendingWrites > 0)
 wait (&r->canRead, &r->sl); /*don't starve
writers */
 while ({r->nActive < 0) /* someone has
exclusive lock */
 wait (&r->canRead, &r->sl);
 r->nActive++;
 r->nPendingReads--;
 spin_unlock (&r->sl);
}
```

# Code for reader-writer locks

```
void unlockShared (struct rwlock
*r)
{
 spin_lock (&r->sl);
 r->nActive--;
 if (r->nActive == 0) {
 spin_unlock (&r->sl);
 do signal (&r->canWrite);
 } else
 spin_unlock (&r->M);
}
```

```
void lockExclusive (struct rwlock
*r)
{
 spin_lock (&r->sl);
 r->nPendingWrtes++;
 while (r->nActive)
 wait (&r->canWrite, &r->sl);
 r->nPendingWrites--;
 r->nActive = -1;
 spin_unlock (&r->sl);
}
```

# Code for reader-writer locks

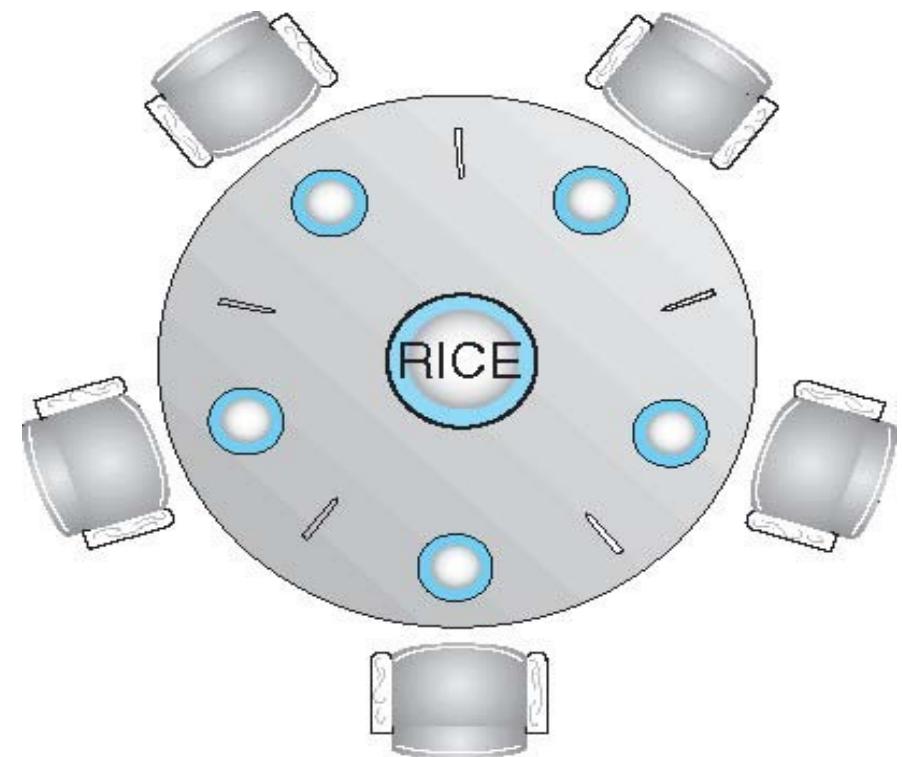
```
void unlockExclusive (struct rwlock *r){
 boolean t wakeReaders;
 spin_lock (&r->sl);
 r->nActive = 0;
 wakeReaders = (r->nPendingReads != 0);
 spin_unlock (&r->sl);
 if (wakeReaders)
 do broadcast (&r->canRead); /* wake
allreaders */
 else
 do_signal (&r->canWrite);
/*wakeupsinglewriter */
}
```

Try writing code for  
downgrade and  
upgrade

Try writing a reader-  
writer lock using  
semaphores!

# Dining-Philosophers Problem

- Philosophers spend their lives thinking and eating
- Don't interact with their neighbors, occasionally try to pick up 2 chopsticks (one at a time) to eat from bowl
  - Need both to eat, then release both when done
- In the case of 5 philosophers
  - Shared data
    - Bowl of rice (data set)
    - Semaphore chopstick [5] initialized to 1



# Dining philosophers: One solution

The structure of Philosopher i:

```
do {
 wait (chopstick[i]);
 wait (chopStick[(i + 1) % 5]);
 // eat
 signal (chopstick[i]);
 signal (chopstick[(i + 1) % 5]);
 // think
} while (TRUE);
```

What is the problem with this algorithm?

# Dining philosophers: Possible approaches

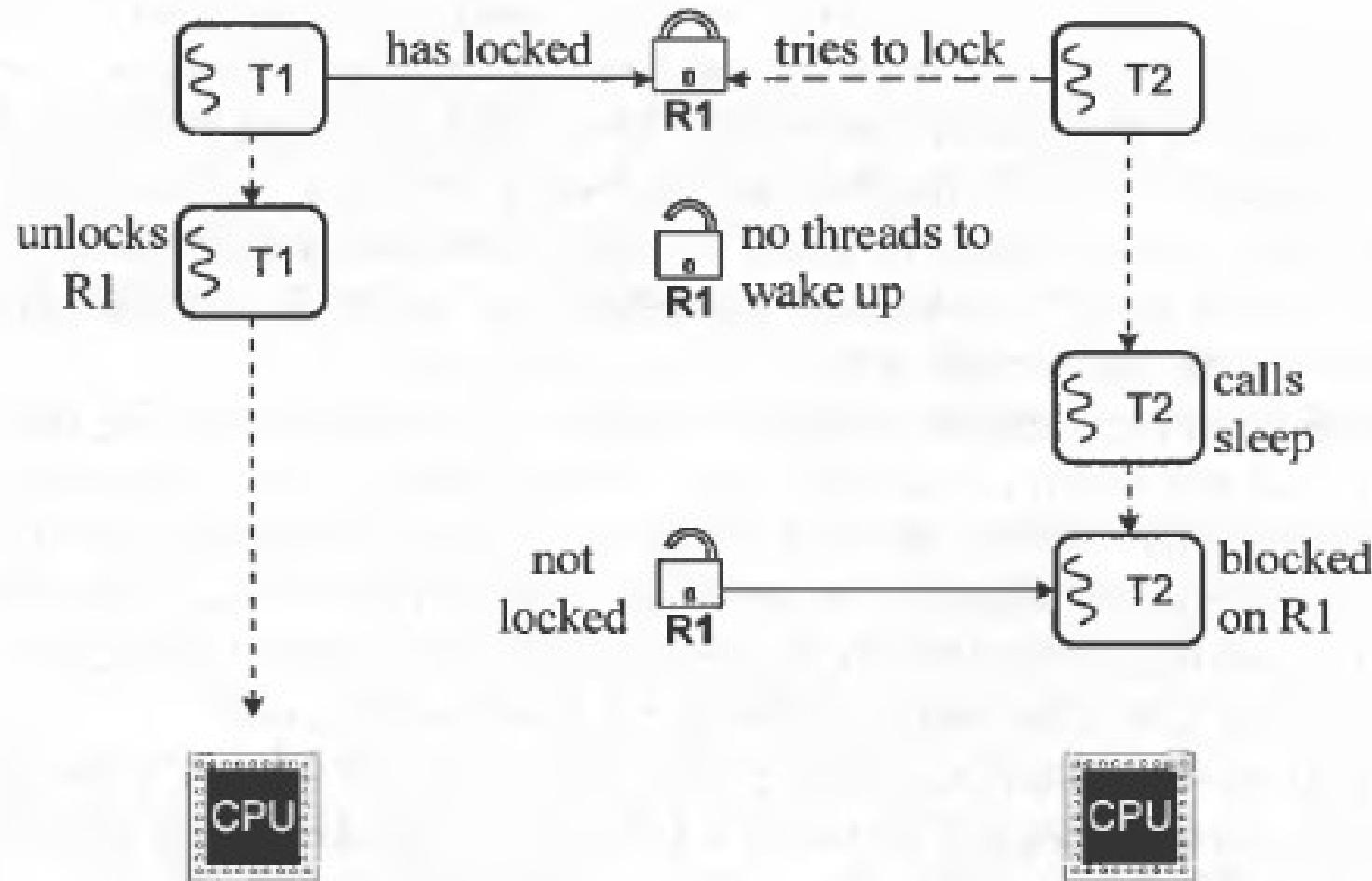
- Allow at most four philosophers to be sitting simultaneously at the table.
- Allow a philosopher to pick up her chopsticks only if both chopsticks are available
  - to do this, she must pick them up in a critical section
- Use an asymmetric solution
  - that is, an odd-numbered philosopher picks up first her left chopstick and then her right chopstick
  - whereas an even-numbered philosopher picks up her right chopstick and then her left chopstick.

# Other solutions to dining philosopher's problem

- Using higher level synchronization primitives like 'monitors'
-

# **Practical Problems**

# Lost Wakeup problem



Requires some mechanism to combine the test for the resource and the call to sleep () into a single atomic operation.

Figure 7-6. The lost wakeup problem.

# Lost Wakeup problem

- **The sleep/wakeup mechanism does not function correctly on a multiprocessor.**
- **Consider a potential race:**
  - Thread T1 has locked a resource R1.
  - Thread T2, running on another processor, tries to acquire the resource, and finds it locked.
  - T2 calls sleep() to wait for the resource.
  - Between the time T2 finds the resource locked and the time it calls sleep(), T1 frees the resource and proceeds to wake up all threads blocked on it.
  - Since T2 has not yet been put on the sleep queue, it will miss the wakeup.
  - The end result is that the resource is not locked, but T2 is blocked waiting for it to be unlocked.
  - If no one else tries to access the resource, T2 could block indefinitely.
  - This is known as the lost wakeup problem,
- **Requires some mechanism to combine the test for the resource and the call to sleep() into a single atomic operation.**

# Thundering herd problem

- **Thundering Herd problem**
  - On a multiprocessor, if several threads were locked the resource
  - Waking them all may cause them to be simultaneously scheduled on different processors
  - and they would all fight for the same resource again.
- **Starvation**
  - Even if only one thread was blocked on the resource, there is still a time delay between its waking up and actually running.
  - In this interval, an unrelated thread may grab the resource causing the awakened thread to block again. If this happens frequently, it could lead to starvation of this thread.
  - This problem is not as acute on a uniprocessor, since by the time a thread runs, whoever had locked the resource is likely to have released it.

# **Case Studies**

# Linux Synchronization

- Prior to kernel Version 2.6, disables interrupts to implement short critical sections
- Version 2.6 and later, fully preemptive
- Linux provides:
  - semaphores
  - spinlocks
  - reader-writer versions of both
  - Atomic integers
- On single-cpu system, spinlocks replaced by enabling and disabling kernel preemption

# Linux Synchronization

- **Atomic variables**

- atomic\_t** is the type for atomic integer

- **Consider the variables**

- atomic\_t counter;**

- int value;**

| <i>Atomic Operation</i>        | <i>Effect</i>          |
|--------------------------------|------------------------|
| atomic_set(&counter,5);        | counter = 5            |
| atomic_add(10,&counter);       | counter = counter + 10 |
| atomic_sub(4,&counter);        | counter = counter - 4  |
| atomic_inc(&counter);          | counter = counter + 1  |
| value = atomic_read(&counter); | value = 12             |

# Pthreads synchronization

- Pthreads API is OS-independent
- It provides:
  - mutex locks
  - condition variables
- Non-portable extensions include:
  - read-write locks
  - spinlocks

# **Synchronization issues in xv6 kernel**

# Difference approaches

- **Pros and Cons of locks**
  - Locks ensure serialization
  - Locks consume time !
- **Solution – 1**
  - One big kernel lock
  - Too inefficient
- **Solution – 2**
  - One lock per variable
  - Often un-necessary, many data structures get manipulated in once place, one lock for all of them may work
- **Problem: ptable.lock for the entire array and every element within**
  - Alternatively: one lock for array, one lock per array entry

# Three types of code

- **System calls code**
  - Can it be interruptible?
  - If yes, when?
- **Interrupt handler code**
  - Disable interrupts during interrupt handling or not?
  - Deadlock with iderw ! - already seen
- **Process's user code**
  - Ignore. Not concerned with it now.

# Interrupts enabling/disabling in xv6

- **Holding every spinlock disables interrupts!**
- **System call code or Interrupt handler code won't be interrupted if**
  - The code path followed took at least once spinlock !
  - Interrupts disabled only on that processor!
- **Acquire calls pushcli() before xchg()**
- **Release calls popcli() after xchg()**

# Memory ordering

- Compiler may generate machine code for out-of-order execution !
- Processor pipelines can also do the same!
- This often improves performance
- Compiler may reorder 4 after 6 --> Trouble!
- Solution: Memory barrier
  - `__sync_synchronize()`, provided by GCC
  - Do not reorder across this line
  - Done only on acquire and release()

- Consider this

- 1) `l = malloc(sizeof *l);`
- 2) `l->data = data;`
- 3) `acquire(&listlock);`
- 4) `l->next = list;`
- 5) `list = l;`
- 6) `release(&listlock);`

# Lost Wakeup?

- **Do we have this problem in xv6?**
- **Let's analyze again!**
  - The race in `acquiresleep()`'s call to `sleep()` and `releasesleep()`
- **T1 holding lock, T2 willing to acquire lock**
  - Both running on different processor
  - Or both running on same processor
  - What happens in both scenarios?
- **Introduce a T3 and T4 on each of two different processors. Now how does the scenario change?**
- **See page 69 in xv6 book revision-11.**

# Code of sleep()

```
if(lk != &ptable.lock){
 acquire(&ptable.lock);
 release(lk);
}
```

- Why this check?
- Deadlock otherwise!
- Check: wait() calls with ptable.lock held!

# Exercise question : 1

Sleep has to check lk != &ptable.lock to avoid a deadlock

Suppose the special case were eliminated by replacing

```
if(lk != &ptable.lock){
 acquire(&ptable.lock);
 release(lk);
}
```

with

```
release(lk);
acquire(&ptable.lock);
```

Doing this would break sleep. How?

`

# bget() problem

- **bget() panics if no free buffers!**
- **Quite bad**
- **Should sleep !**
- **But that will introduce many deadlock problems. Which ones ?**

# **iget() and ilock()**

- **iget() does no hold lock on inode**
- **Illock() does**
- **Why this separation?**
  - Performance? If you want only “read” the inode, then why lock it?
- **What if iget() returned the inode locked?**

# Interesting cases in namex()

```
while((path = skipel(path, name)) != 0){
 ilock(ip);
 if(ip->type != T_DIR){
 iunlockput(ip);
 return 0;
 }
 if(nameiparent && *path == '\0'){
 // Stop one level early.
 iunlock(ip);
 return ip;
 }
 if((next = dirlookup(ip, name, 0)) == 0){
 iunlockput(ip);
 return 0;
 }
 iunlock(ip);
 ip = next;
}
--> only after obtaining next from
dirlookup() and iget() is the lock
released on ip;
-> lock on next obtained only after
releasing the lock on ip. Deadlock
possible if next was “.”
```

Xv6

Interesting case of holding and releasing  
ptable.lock in scheduling

**One process acquires, another releases!**

# Giving up CPU

- A process that wants to give up the CPU
  - must acquire the process table lock ptable.lock
  - release any other locks it is holding
  - update its own state (proc->state),
  - and then call sched()
- Yield follows this convention, as do sleep and exit
- Lock held by one process P1, will be released another process P2 that starts running after sched()
  - remember P2 returns either in yield() or sleep()
  - In both, the first thing done is releasing ptable.lock

# Interesting race if ptable.lock is not held

- Suppose P1 calls `yield()`
- Suppose `yield()` does not take `ptable.lock`
  - Remember `yield()` is for a process to give up CPU
- Yield sets process state of P1 to **RUNNABLE**
- Before `yield`'s `sched()` calls `swtch()`
- Another processor runs `scheduler()` and runs P1 on that processor
- Now we have P1 running on both processors!
- P1 in `yield` taking `ptable.lock` prevents this

# Homework

- **Read the version-11 textbook of xv6**
- **Solve the exercises!**

