HW 4

Dustin Badillo
3/13/19
CSE 460

HW

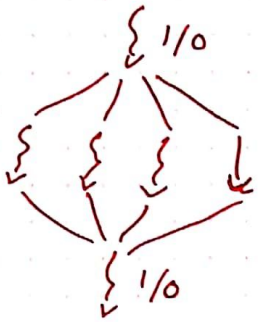
4.81) a) registers not shared
Heap not shared
global shared
stack not shared

registers not shared
instructions (program) shared
Devices "
Files shared
semaphores shared
pipes shared

stack and program counter are not shared

base/limit not shared

4.48)
4.14) Two dual core processors => 4 cores
single file must be read/written onto
# of threads for I/O? 1
# of threads for cpu intensive portion? 4 cores



4.16) Linux does not distinguish between threads and processes, everything is a task
windows: process contains pointers to threads belonging to it.

clone()
PCB TCB

task_struct
does not have PCB or TCB
alot of sharing - thread
little sharing - program process

~~First~~ in linux a thread is a process that shares resources with other processes,
no concept of pure threads (threads within the same process)
in linux all tasks get the same time slice
in windows has the flexibility of dividing a process time slice among its threads
also its easy to identify pure threads

Advantage to linux approach: easier to implement
Both linux and windows use 1-to-1 model (user and kernel level mapping of threads)

P1 - P2 ✗ P2 - P3 ✗ P1 - P3 ✓

semaphore wrt = 1, mutex = 1, integer readcount = 0

```
P1
wait(mutex)
    readcount ++
    if (read count > 0)
        wait(wrt)
signal(mutex)
C.S.
wait(mutex)
    readcount --
if (read count == 0)
    signal(wrt)
signal(mutex)
```

```
P2:
wait(wrt)
C.S.
signal(wrt)
```

P3: // exact copy of P1

Philosophere problem: Philosophers i (for $1 \le i \le 4$)

```
wait(chopstick[i])
wait(chopstic[(i+1)%5])
"eat"
signal(chopstic[i])
signal[chopstick[i]]
"think"
```

wrong →

answer is Deadlock

```
Phil 0:
Wait(chopstick[4])
Wait(chopstick[0])
eat
signal(chopstic[i+1])
signal(chopstic[i])
"think"
```

Round Robin allows for interaction

Scanned by CamScanner

CSE 460
3/11/19 Pg 1

HW

not crit
at same time

P1 —✗— P2 —✗— P3

ok for P1/P3
to be crit at same time

Semaphore S=1, t=1

Does not work

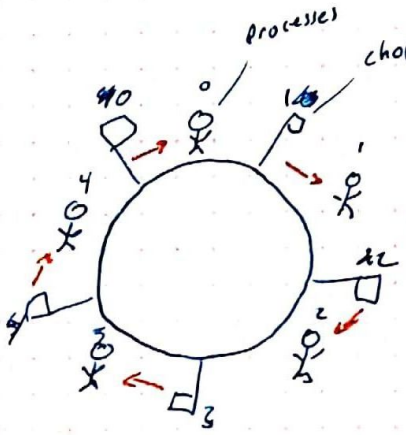
P1	P2	P3
wait(s)	wait(s)	wait(t)
	wait(t)	
	cs	cs
	signal(t)	
signal(s)	signal(s)	signal(t)

make P1 and P3 readers
P2 writers

Dining philosophers

5 right handed
1 left handed

can have only 4 philosophers at the table at a time and have a wait on 4 phil



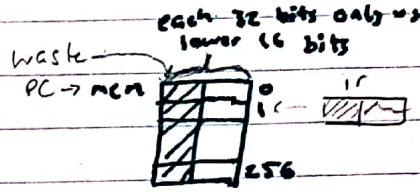
1st attempt (not right)
semaphore chopsticks[5] = {1, 1, 1, 1, 1}
Philosopher i :
wait (chopstick[i])
Blocks each other here → wait(chopstick[(i+1)%5]) // pick up chop
"eat"
signal (chopstick[i])
signal (chopstick[(i+1)%5])
"think"
loop back

Deadlocks
circular wait

Scanned by CamScanner

1/7/19
Pg 1
CSE 460
Operating systems

PC - (address to the next instruction program to execute starts at 0 first instruction
ir - instruction you want, to register you bring it in
(from memory) instruction you want to execute
sr - if there is a error, status
sp - stack of activiction record, call and return
every time there is a function call there is a push
on the stack of activation record
Clock - keeps track of clock ticks
ALU - basic arithmetic
mem - 16 bits 256 words
vector <int> mem (256);



assign 0 to don't care
don't care = 0

instruction formats

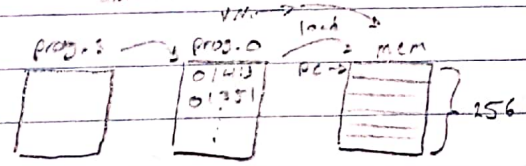
d . . . . d V L E G C } ir

format 2
op code
operation
immediate instruction: carrys one of the operands

OP	RD	I	RS	unused
15:11	4 bits	8	7:4	3:0

op RD I ADDR/const

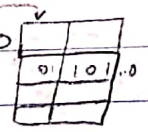
add: add carry
after you perform the operation you set the new value of carry



add 1 2 // content of reg 2 and add into reg 1

0xffff
f 15 16 1's
r[0] &= 0xffff
16 16
r[0] 0110 1 ...10
0xfff 0....00 1.11.11 &
0...00 10

0
zero out spill over bits



in sp return means pop

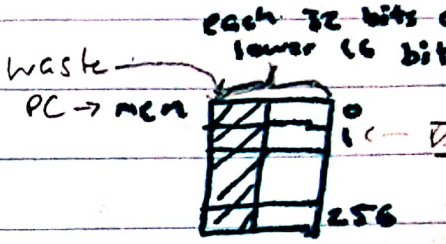
Scanned by CamScanner

Clock - keeps track of clock ticks

ALU - basic arithmetic

mem - 16 bits 256 words

vector <int> mem (256);



to do-t core

core = 0

n formets

d . . . . d V L E G C } ir

over flow

less equal greater carry

Format 2

operation (add/load ex.) → op code

OP	RD	I	RS	UNUSED
15:11	2 bits since 4 reg	8	7:6	5:0

reg dest 10:9

immediate 0

reg source

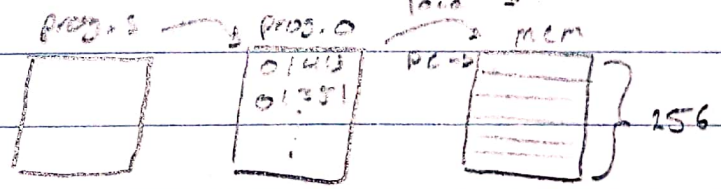
truction:
of the operands

OP RD I ADDR/const

assembler - assembler, assembly language to object program

carry

m the operation
value of carry

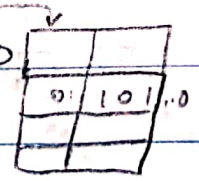


add 1 2 //content of reg 2 and add it to reg 1

16 is

xffff
16
. . . 10
11 11 &
10

0
zero, it spill over bits



in sp return means pop

Scanned by CamScanner

$\not{b}$ = blank

CSE 460
1/9/A Pg 1

Random notes

os.cpp

array

argument value

argument count
argc = 3

$\not{b}$ | os | prog.s | -o | arg.v

0 1

arg[0] = "os"

to grab ~~first~~ sec. arg value int main (int argc, char *argv[1])

open file / call function / close file

String name = assembly file.substr (0, pos); // reads the prog file name from 0; pos

assembly file converts to .s to .o

if (as.assemble (assembly, object code)

where we use our assembler

in file

out file

assembly.close ();

object code.close ();

open the new object file

object code.open (object file.c_str (), ios::in);

returns out of the string

assembler.h

2 functions

assembler.cpp

comes in from input

getline (in, line)

places

only time reading from file

input

while !eof

input

}

at end it builds the instruction and places it to out string and loops until end of file

continue and break are almost the same

right after while it checks if file is empty

if there is a legitimate line

stringstream str (line.c_str());

input

str >> opcode;

5 colums

setfill ('0') << setw(5) << instructions << endl;

| -> bitwise or / same

instr = instr + rd << 9 + addr;

Scanned by CamScanner

PC starts here

We have main is starting point since we can have functions everywhere

D4221126

CSE 466
1/9/19
PS C

Virtual machine.h

main function is void run(fstream&, fstream&, fstream&);

places in memory

Virtual machine.cpp. where you worry about carry bit

base = 0 ← start

for (limit = 0; objectcode >> mem[limit]; limit++);

ir = mem[PC]; //fetch ✱ fetch -> increment cycle

PC++ //increment PC PC goes all the way to the limit

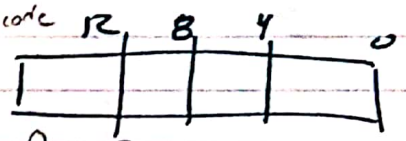
// execute

bitwise and order it to get 0's opcode

opcode = (ird 0xf800) >>11;

0's before 6

rd = (ir & 0x600) >>9; → 6



1100 0000 0000

constant & 0x80 ← sign extention 4's four 1's

1000 0000

checking for carry bit in case of a negative value sign extend

8 0
1000 0000

if (constant & 0x80) constant |= 0xff00;

2 pos = neg
2 neg = pos } overflow

compare sign bit with result and check for overflow

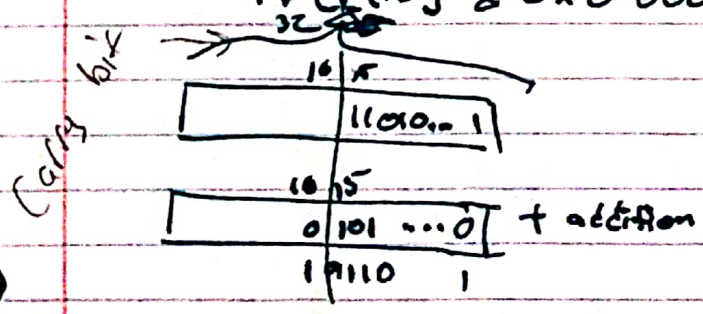
int sign1 = (r[rd] & 0x8000) >> 15;

int sign2;

to perform + on the machine we sign extend

checks if sign is 1 if so sign extend

if (r[rd] & 0x8000) r[rd] |= 0xffff0000;



e = 1110 fail 1110

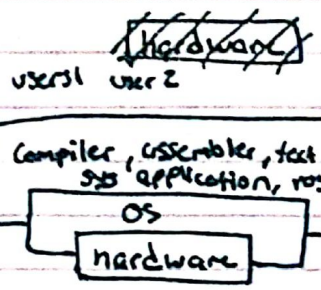
else sr & = 0xffe

ch 1 intro

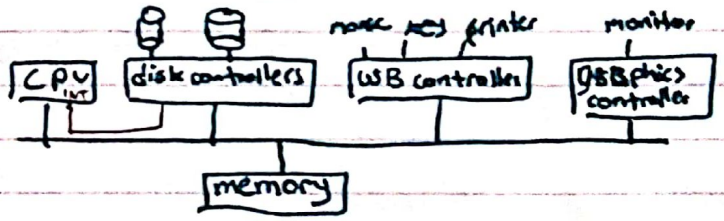
O.S.

- Program, a resource manager and allocator
  - ↳ cpu time, memory, I/O devices, ...
- Provides a convinient enviroment for users to execute programs efficiently
- interface layer between users / application programs and hardware

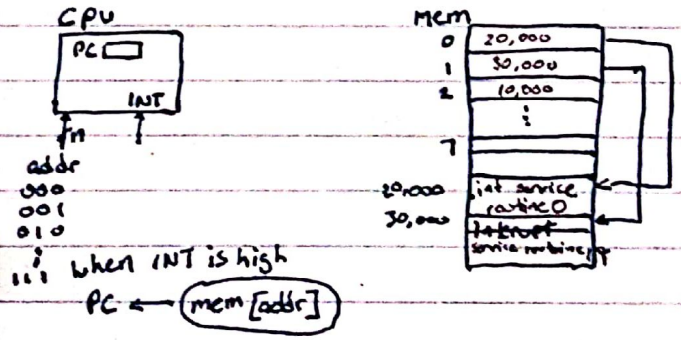
Fig 1.1



Computer system organization
Fig 1.2



- hardware interupts cpu using signals
  - interupt is a signal to the cpu saying something is done
- software interupts / traps the OS using system calls
  - ↓ API to OS / way to communicate with OS
  - "when you open a file that is a system call to the OS



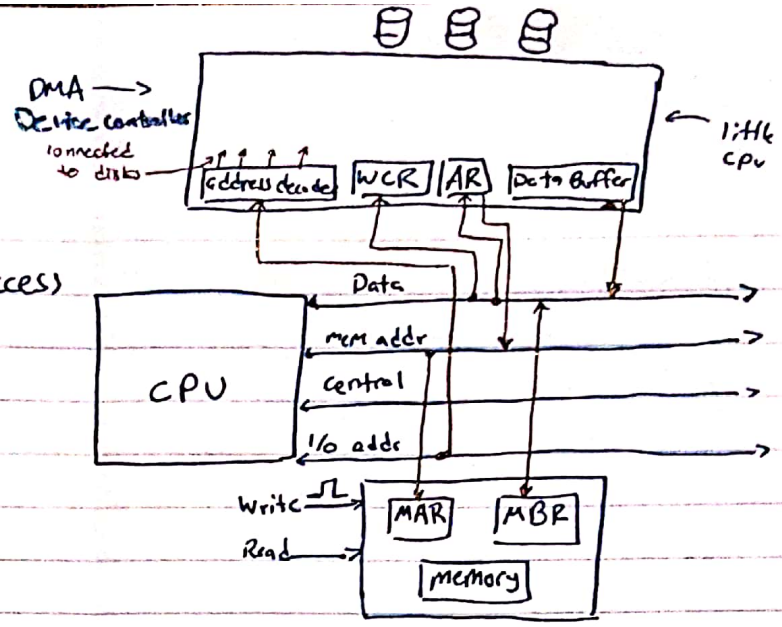
Device Drivers	vs	Device controllers
software		hardware
Part of OS (API for controller)		buffer (mes buffer)
"understands controller"		

Scanned by CamScanner

CSE 460
Pg2 1/14/19

DMA - Direct memory Access
Fig 1.5

MAR - mem addr reg
MBR - mem buf reg
AR - addr register



input operation

(control lines)
1. CPU signals DMA to initiate I/O operation
2. CPU supplies device # (Addr Decoder), where to read fro-
   memory address where to go
   # of bytes to transfer (WCR) how many bytes
3. DMA MAR ← AR
   reads from device
   transfers one byte from Data Buf to MBR
   memory performs a write
   DMA increments AR, decrements WCR
   repeat until WCR = 0
4. DMA interrupts CPU saying DMA op is complete

Cycle Stealing

instr fetch-execute cycle
instr fetch (mem)
decode
operand read (mem)
execute
write back (mem)

DMA can only steal cycles

Scanned by CamScanner

Sec 1.4
O.S. Structure

Prog, control cards, input, output

Batch System : Similar jobs were batched together
- OS would transfer control from one job to another

Multi-Programing
Fig 1.9

- increase cpu utilization % of time busy
- overlap cpu execution of one job with I/O of another job
- Job scheduling among ready jobs which one execute next

Time Sharing (Multitasking) - extend multiprograming
- cpu switches among jobs in a round ~~t~~robin fashion and time-~~slicing~~ slice allocation
  10 m sec
- Each user gets the impression of being the sole user on the system
- This provides an interactive system Virtual memory

Scanned by CamScanner

1/16/19
CSE 460
Pg 1

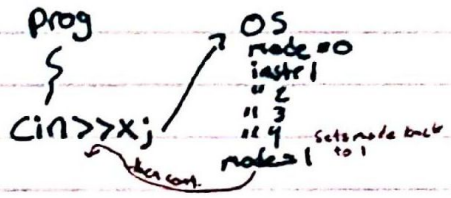
Interrupts

OS is interrupt driven
- an interrupt is an event/request for service coming from I/O device, user, processes
- Software interrupt (trap, exception)
- dual mode operation provides protection from one process to another
  - supervisor mode
  - Privileged instructions

Mode

mode is a hardware register
- only OS sets it (supervisor or nonsupervisor) 0 1
- mode=0 supervisor mode = 1 user mode

Fig 1.10



Instructions

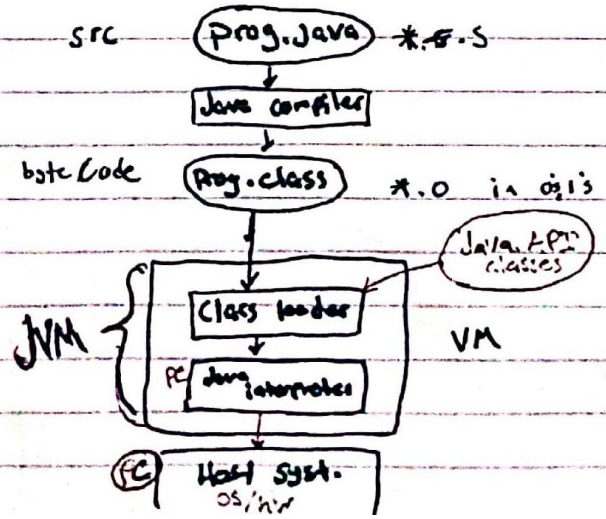
do not perform priviledged instructions in user mode

instructions are separated into two broad categories
- Priviledged instructions flag=1
- non-Priviledged instructions flag=0

priv instr flag	mode	
0	0	✓
0	1	✓
1	0	✓
1	1	X ← no priv instr in user mode (catch this with a and gate)

PART of pg 3 1/16/19

Java virtual machine (Sun 1995)



from hardware
↓
interpreter is a program that runs other program

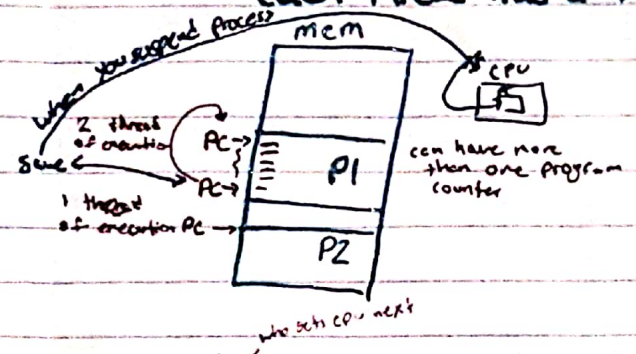
11

1/15/19
CSE 460
B2

1.6 Process Management

Process
needs program counter

a running program
- an active entity, a program is a passive entity
- may have multiple threads of execution (≥1 PC)
  - each thread has a PC



each CPU can only execute one thing at a time

PCB: Process Control Block
- contains the PC

each process has its own PCB
- as soon as a ~~program~~ process is created it has a PCB
- a process is a program that has a PCB

OS
OS decides which process is next

1. schedule process and threads on CPU(s).
2. suspends and resumes processes / threads
3. Performs process / thread synchronization and communication

1.7 memory management

memory : large array of bytes
- A portion of the program must reign in memory in order to execute

OS

1. keeps track of which part of memory is allocated to what process
2. decides which process to move into and out of memory
3. Allocates and deallocates mem (new / delete) (heap) c++

as you do more allocation heap gets bigger

when they meet you are out of memory

every function call builds the stack, return removes

Scanned by CamScanner

1.8 Storage management

- Mapping files to secondary storage
- system calls to (storage management)
  - create / delete files or directories
  - open / close " "
  - read / write " "

a file that has pointers to other files

other system calls for non storage management

- System call fork() : create duplicate process
  - exec() : change its image
  - malloc() : memory allocation ex. new

Computing enviroments

- A client server
  - main() - client class - server

  main makes request and class executes it
- A peer - to - peer network of services

  ~~server - client~~ ~~client~~ ~~client~~

  Peer — Peer — Peer

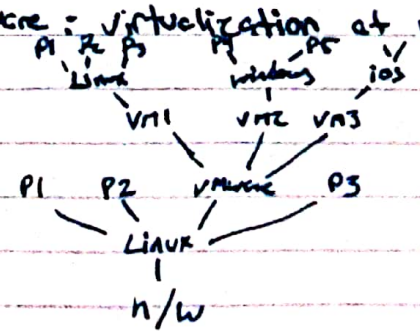
- Mobile computing
- Virtual Machine (IBM VM370 / 1972)
  - abstract out hardware (hw) of a computer into different execution enviroments giving the illusion of multiple computers

Fig 1.2

P6	P7	P4	P5	P1	P2	P3
OS1		OS2		OS3		
h/w		h/w		h/w		
virtualization						
h/w						

VMware

VMware : virtualization at user mode



Scanned by CamScanner

OS Structures

From POV: of
1. Service it provides
2. its user interface
3. its components

2.1 Services

Fig 2.1
- program execution, (I/O: operation, effency, protection)
- Communication
- File system
- Protection and security

2.2 User-OS interface

- Command interpreter - shell
- GUI - Xerox PARC 1973 ex. Mac OS, Windows

2.3 System calls

Provides portability ⟶ about / system call

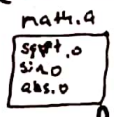
- interface to OS services Fig 2.5
- usually there is a interface API on top of system calls
  - Ex. CreateProcessX() → NTCreateProcess() ← (system call)

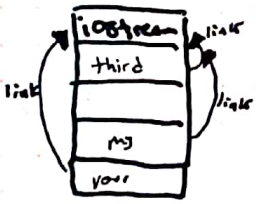
Linking

g++ kernal.o newfilers.o ← 2 ways static link, dynamic link

g++ main.cpp → main.o
The inputs are executed by iostream

link together (static linking)
(.a in unix) ar (archive command to create .a)

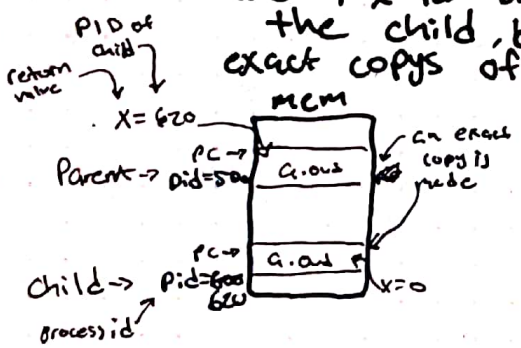




link to the same file so them (dynamic) share are not more then one copy
(.dll in windows) + table needed
(.so in unix shared object)

fork.cpp

Value of x for the Parent is the Process ID of the child, both the child and parent are exact copys of eachother



PS -ef process list command

Tamoe River

Scanned by CamScanner

Layers

1. shell (Bash)
2. Virtual memory (Paging)
3. Device drivers
4. memory management
5. Process management
6. hardware

Chapter 3 Processes

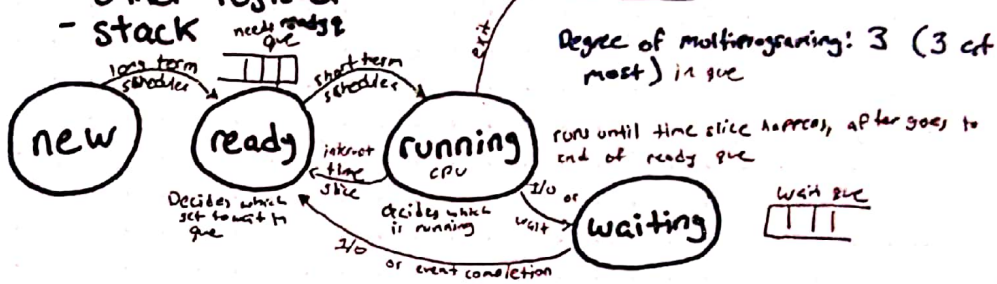
- A Process is a running program - an active entity
  - PC (Program counter)
  - other register
  - stack

make a process

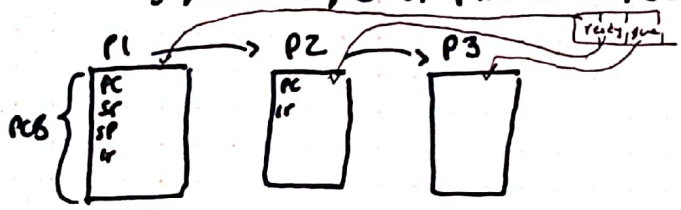
Fig 3.1

Fig 3.2 (heart of Prog) process state

PHASE 2



- ready que is a que of pointers to pcb



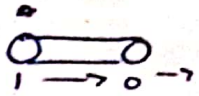
- A process is represented by process control block (PCB)

Fig 3.3 what PCB includes

1. PC
2. other CPU register
3. base limit, page table (page table)
4. Process state, process id /name
5. accounting information; cpu time, throughput
6. CPU scheduling, priority
7. I/O status

- sheduling is overhead (minimize it)
- Context switch refers to multiplexing CPU among processes Current Process status is saved (in its PCB) and a new Process status (from its PCB) is loaded in CPU Fig 3.4

Scanned by CamScanner



Pipe.c

~~PIPE~~ pd[1] ~~and~~ write to
pd[0] read from

parent and ~~child~~ child have access to both ends of the pipe

char y // 1 byte buffer

address, how many bytes to write

if (write(pd[1], ~~&~~y, 1))

Fork

the parent returns the process id of the child

path, name, ~~parameters~~ from pwd/home

exec 3.1.c

execl ("bin /ls", "ls", Null) will replace entire ~~factor~~ file with what is inside of ls

$/ls -l

... "ls", "-L", Null)

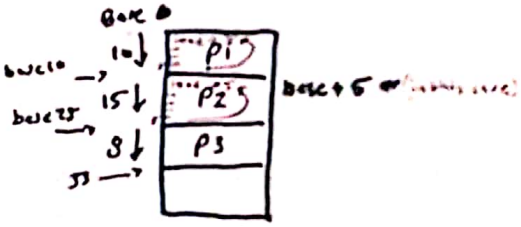
fork → exec("ls") → "ls"

execlp ("ls", "ls", null) ((works to put searches through your path

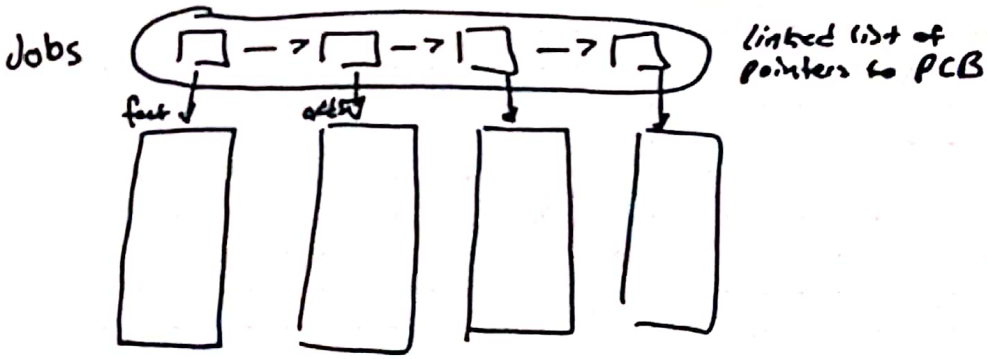
fork better for simple programs if you dont want to change the program
exec better for manipulation of the files

Structure vs class
default public
default private

~~daemon~~ daemon - process that runs in the background



Contest switching Fig 3.4



* all code in OS.cpp minimal changes in virtual machine *

when sp = 256 // the stack is empty

system("ls *.s > progs"): grab all .s files and place it in progs

loop (list < PCB* > jobs;
PCB* p = new PCB;) loop for each program in progs.

Scanned by CamScanner

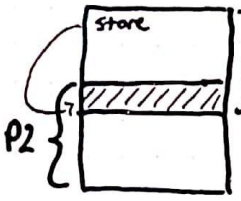
Utilization: amount of time busy
Throughput: amount transfered per sec

CSE 460
2/4/19 pg 1

3.4 IPC for cooperating Processes

cooperating processes: 1) communication 2) synchronization
- we need this for
  - information sharing ex. shared file
  - increasing speed (computation speedup)
    - Break tasks into subtasks, run in parallel
  - modularity - create better organization
  - ~~convenience~~ convenience: some user edditing, compiling, printing at the same time

IPC can be done through (Fig 3.12) message passing or shared memory

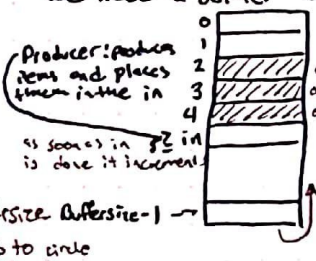


sends messages using OS

Ex. Producer - consumer Problem

Producer	Consumer
Print prog	printer driver
webserver	web browser
compiler	assembler
assembler	loader
⋮	⋮

we need a buffer between producer/consumer



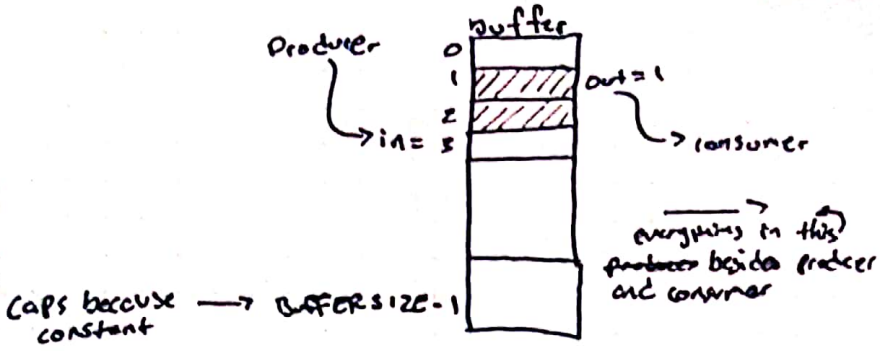
in = (in+1) % Buffersize Buffersize-1 → when end is reached circle back to 0

↑ this is to circle to top

code for producer on pg 125
Consumer on pg 126

* in the initial case in and out are both at 0 * if both are equal then buffer is 0
- as soon as in is 1 space behind out stop it from producing

Scanned by CamScanner




```
#define BUFFER_SIZE 10
class Item {
  ==
};
Item buffer[BUFFER_SIZE];
int in=0;
int out=0;
```

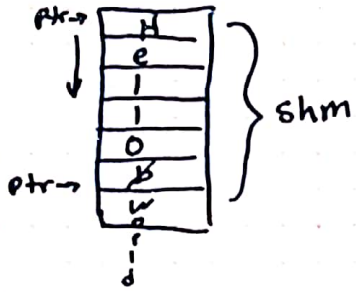
Change mode

chmod

```
$ls
--file1---
chmod 044 file1
```

	me	group	everybody
~~rwx~~	rwx	rwd	rwx
	110	100	100
	6	4	4
	111	110	100
	7	6	4

```
ptr = mmap(0, size, prot_write, map_shared, shm_fd, 0);
// you get a pointer to
```




```
printf("%s", (char *) ptr);
```
reading from pointer

[redacted]

```
item = 123;   // inputs it 123 and then increments
ftruncate(segment_fd, (BUFFER+2 * 4));   //resize
```
size of buffer 52; 2 int; 4 bytes

```
srand(tv.tv_used);   // set seed for random #
```
time; microseconds

```
wait(0)   //waits for child
```
needs to wait for child since parents need to live longer than ~~parents~~ children

Message Passing

sent (message)
recive (message)

fixed vs. variable sized messages
↳ easier to implement in OS
→ easier to program and use for programmer

Direct vs. indirect
↳ explicitly naming of sender and ~~sender~~ reciever
↳ symmetric or Asymmetric

Q: send(P, mess)
P: recv(Q, mess)

Q: assend(P, mess)
P: recv(id, mess)
recv from any process in this case id = Q

→ message is sent to mail box or ports (mail box is buffer)

Q: send(A, mess)
P: recv(A, mess)
P and q must share mail box A, A can be owned by OS or processes using it. if on 2 diff machines then OS has it

MPI: Message Passing interface:
send: non Blocking
recive: Blocking

Synchronization
~~Blocking or Non Blocking~~
~~Blocking~~

Blocking send "recv" vs non Blocking send "recv"

Scanned by CamScanner

3 chap 3 1 chap 1 1 chap 2 5 ε total

Chap 3 Fork, shared memory, process management (PCB) Fig (3.2) Fig (3.4) PCB Z context switching

3.8 Process management (PCB) Fig (3.2) Fig (3.4) Fig (3.3)
- to execute a program the pc of that program has to be on the hardware
- the : in ~~the~~ 3.4 the pc ~~is~~ of the OS is running on the hardware

chap 2 OS Design

MPI - messese passing interface
argc - argument count
char **argv - array in an array "jessed end array"

MPI: np is the number of processes
my_rank: process # PID original process does the "else"
null terminating char strlen(message) +1
MPI_comm_world: MPI_init created it

Socket
(on same machine)
only programing language

~~IP~~ IP number which specifies a machine in a network and an integer (port)

Socket (IP#, Port#)

ex. → 139.182.100.05 integer (0,-1024) ← reserved for system
1025+ free

$4x^{2-1} =$

$2(4) = 8x^1$

x^2 $2x^{2-1} \rightarrow 2(2)x^1$

$4x$

mut add
sub div

$\int 4x^2$

$\frac{1}{n+1}x^{n+1} =$

$\frac{1}{4+1}x^{2+1} = \frac{1}{5}x^3$

$x^{2-1} = 2x^1$

$2x^1$

$4x^2$

$\frac{4x^3}{3}$ $\frac{3(4)x^2}{3}$

$3x^{3-1} = 3(3)$
$9x^2$

$3x^3 = 9x^2$

$4x^2$

$36x^5$

$3x^3$ $\frac{3x^4}{4}$

$\frac{3x^4}{4}$ $\frac{(4)(3)x^3}{4} = 3x^3$

Scanned by CamScanner

-g to compile with debugging

gbd gnu debuger

set break point at line you want to see variables

- break file_name: line
- run prog.s

p for print then outputs answer

- p pc

c for continue

- c

n to go to next line

- n

help to bring up guide

- help
- quit "to exit

RPC - remote procedure call

read chap 8 and 9

- we need virtual memory because if memory is small and we want to run something larger we can use virtual memory,
- you bring in pages of your program as needed. ex. if we have 4 mb of memory you break it up into 4 frams. lets say we have a program that, ⊙

- page table is a simple vector

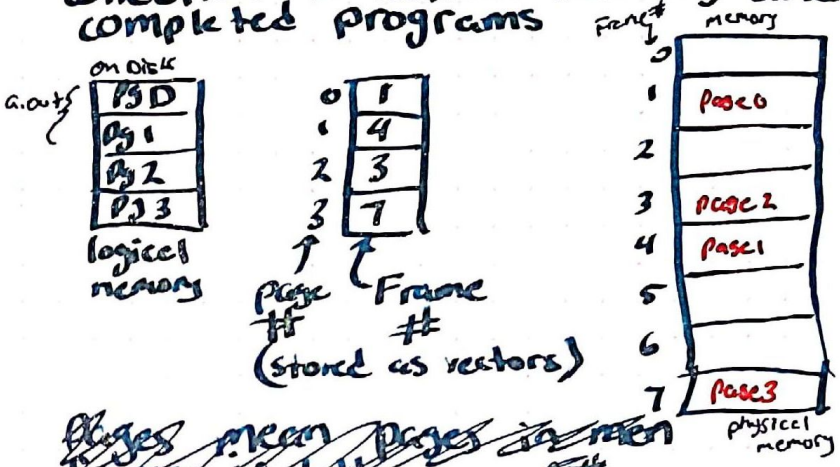
Scanned by CamScanner

external Fragmentation

when there is not enough space in memory to run the program, need garbage collection to restruct memory and remove completed programs

Fig 8.11

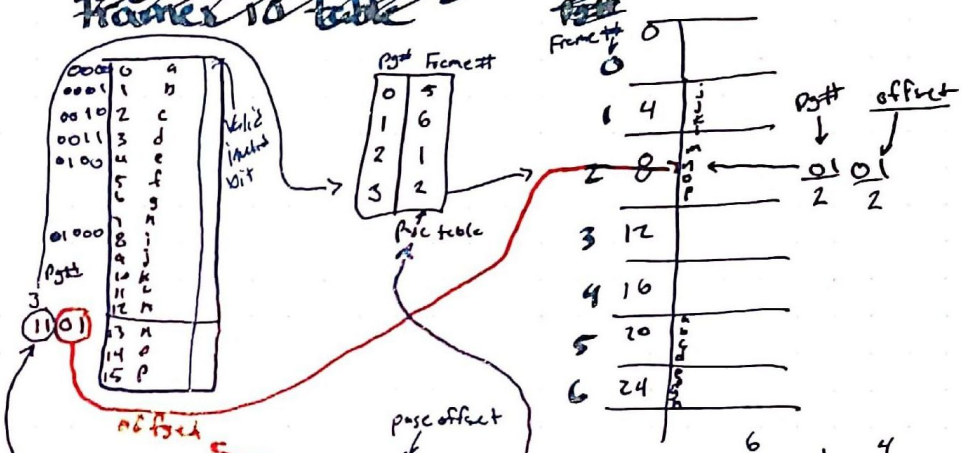
Frame and page size the same



you only bring pages that you need in memory

Frame-

~~Pages mean pages in mem frames in table~~



every page table has a valid or invalid bit

offset

page size $= 2^{\text{page offset}}$ bytes

ex.

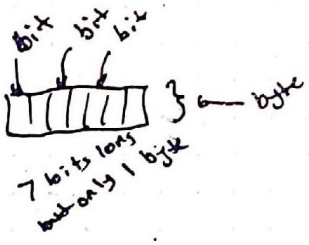
6	4
P#	disp

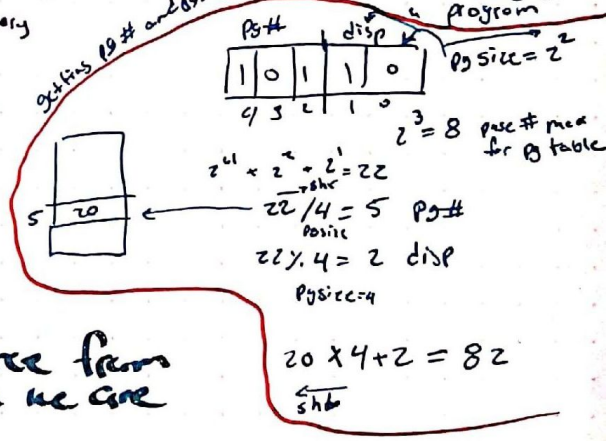
$2^6 = 64$ pages

$2^4 = 16$ bytes with 16 bytes in it

$2^{10} = 1024$ bytes which is max size of program

grab page # from program then go to page table and get the frame # then you go to the program and get the offset and that is where it is stored in memory





internal fragmentation is when there is wasted memory in the page, program does not fill all of last page

need to maintain a free from linked list for when we are adding new processes

TLB

Fig 8.14

Translation Lookaside buffer: in associative memory/ content admissible memory/Map, is another way to store page table to get frame #, the search happens as one parallel search to find page # and frame #

Scanned by CamScanner

Demand Paging

inverted page table

Page replacement algorithm

Pg / frame size = 1024B = 1kB
Logical addresspace : 64kB ← largest possible program
Physical memory : 16 kB ←
max num of pages in a prog? $\frac{64\,kb}{1\,kb} = 64$ pages
# of bits in a logical address? 6
6 bits $2^6 = 64$

Max Num of page table entries? 64
# of bits in each entry? 4 $2^4 = 16$
? → # of bits in physical memory: 14 ← 4+10

4 bits
$2^4 = 16$

by looking you have 16 frames

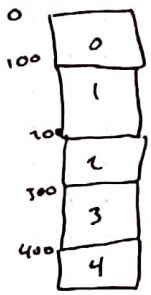
if log addr = 0010101000000011

1000 | 0000000011

4B frames
00 - 10
00
01 4B
10 but
11 2 spaces (disp)

$2^3 = 8$ frames
8 frames
4 B frame
frm : disp
$2^5 = 32$

0	10
1	5
⋮	
9	1
10	8
11	15
01	3
02	6



Suppose 100 B per page, then for the following address sequence
100, 432, 10, 612, 102, 103, 50, ...
we have the following reference string (which pages are accessing)
1, 4, 0, 6, 1, 1, 0

pf = page fault

First-in first out with 3 frames (FIFO) Fig 9.12
record time when frame came in

7 0 1 2 0 3 0 4 2 ...

7	7 / 0	7 / 0 / 1	2 / 0 / 1	2 / 0 / 1	2 / 3 / 1	2 / 3 / 0	4 / 3 / 0	4 / 2 / 0
pf	pf	pf	pf	no pf (already there)	pf	pf	pf	pf
7	0 7	1 0 7	1 0 7	1 0 2	1 3 2	4 3 2	1 3 4	0 2 4

code this wins a que
first in the que is the victim

LRU: Least recently used
record time every time refrenced

(Belady's Anomoly)

7	7 / 0	7 / 0 / 1	2 / 0 / 1	2 / 0 / 1	2 / 0 / 3	2 / 0 / 3	4 / 0 / 3	4 / 0 / 2
pf	pf	pf	pf	no pf (0 was just used)	pf			
7	0 7	1 0 7	2 1 0	0 2 1	3 0 2	0 3 2	4 0 3	2 4 0

use a semi stack
a stack with holes on bottom
bottom of stack is victim
to code this every time a pg is referenced record time and when brought in

have VM take care of counting

frame registers

mem

frame reg ← can do both FIFO and LRU

0	
1	
2	
3	

clock++
all same

Scanned by CamScanner

Global vs local page replacement

Global: all processes compete for all frames.

More popular → Local: each process keeps its own set of frame

Therashing

- when a process spends more time paging than executing
- Thrashing in local page replacement
  ex. suppose each process gets 4 pgs at most
  - if a process needs 5 pages to execute it will ~~crash~~ therash

- Thrashing in a Global page replacement algorithm
  1. a process starts page falting → lowers CPU utilization
  - low utilization increase degree of multiprograming.
  - more processes → more page faults
  - More PFs = lower CPU util

Demand Paging 9.2 or 9.1?

Swap out / Swap In

TCB - Thread control block, PC, other registers, stack pointer is all you need to maintain, significantly faster than PCB

- To implement in code just move TCB around rather than PCB's around

- Threads separate the notion of execution (PC) from the rest of the definition of the process. each notion of execution is a thread

- A process is a task with at least 1 thread
- Context switch between pure threads is faster because of extensive sharing

To create threads in linux use clone()

Scanned by CamScanner

Ch.5 Process Synchronization

$x = 2$

P0: $\boxed{x \leftarrow x+1}$

P1: $\boxed{x \leftarrow x-1}$

exclusive execution (atomic operation)

critical section

Scennario 1:
P0: fetches value of x in some reg
P1: " " reg1
P0: adds 1 to r0
P1: adds -1 to r1
P0: stores 3 in x
P1: stores 1 in x

solution

P0:
wait()
critical section { $\boxed{x \leftarrow x+1}$
signal()

as soon as P0 is done with critical section it sends a signal to P1

P1:
wait() ← system calls
$x \leftarrow x-1$
signal() ← system calls

AKA P(C)

AKA V(C)

Wait and signal implementation

A solution to C.S. problem must satisfy:
1) Mutual exclusion: no more than 1 process in critical section at a time
2) Progress: If no process is in C.S. and there are no process waiting to get in their C.S., one should get in and only these process can make the decision
3) Bounded wait: no livelock

Bounded wait: when you have more than 2 processes and only a select few are alternating between each other making the rest wait longer to be executed

```
class Semaphore {
  Public:
    Semaphore (int v): value (v){ }
    int value
    list <PCB> l;
  };
  wait (Semaphore s)
   {
    s.value --;
    if (s.value < 0){
          s.l.push_back(this -> pcb);
          block();  //waiting in waitq
    }} //if
  } //wait

void signal (semaphore s)
{
    s.value ++;
    if (s.value ≤ 0){
        remove process P from s.l;
        wakeup(P);
    }
}
```

decrement

signal value

Scanned by CamScanner

initial value

semaphore S = 1; if far more than 1 can carry more process at a time

P1: {	P2: {	P3: {
wait (S);	wait (S);	wait (S);
C.S.	C.S.	C.S.
Signal(S);	Signal(S);	Signal(S);
}	}	}

Bounded Buffer Producer-consumer semaphore mutex = 1, empty = BUF_SIZE, Full = 0, int in = out = 0
Prob (again!)

mutex1 = 1
mutex2 = 1

Producer:
"Produce an item"
wait (empty)
wait (mutex)
{ buf[in] = item
in = (in+1) % BUF_SIZE
signal (mutex)
signal (full)

wait with decrement →

Critical section

signals

Consumer:
wait (full)
wait (~~full~~) mutex2
item = buf[out]
out = (out+1) % BUF_SIZE
Signal (mutex)
Signal (empty)

signals

will cause semaphore to deadlock (deadlock) semaphore S = 1, Q = 1

Just bad code (to fix switch S, and Q inside P1) or vice versa

P0:
→ wait(S)
→ wait(Q)
C.S.
Signal(Q)
signal(S)

is locked

P1:
→ wait(Q)
→ wait(S)
C.S.
Signal(S)
signal(Q)

deadlock

reader/writers Problem (from Database)

first reader-writers prob (readers have Priority)
second " " (writers have Priority)

Semaphore write (wrt) = 1, mutex = 1, int readCount = 0

Writer:
wait (wrt);
"write"
signal (wrt)

readers:
wait (mutex)
readcount++;
if (readcount == 1)
wait (wrt)
signal (~~wrt~~ mutex)
"read"
wait (mutex)
read count--;
if (read count == 0)
signal (wrt)
signal (mutex)

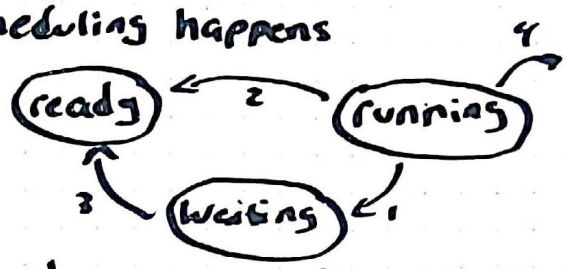
flow test

Scanned by CamScanner

Chap 6 CPU scheduling

multiprograming → CPU scheduling ← short term scheduler

scheduling happens



1,4) non-preemptive : halts

2,3) preemptive : interrupts

Scheduling Criteria

scheduling criteria

CPU-unit utilization: % of time CPU is busy (max) (maximize)

throughput: # of process completed per unit time (seconds) (maximize)

- Bandwith (example)

Process / jobs

time spent in waiting que is ~ idle time

waiting time: time spent in ready que, waiting for CPU (minimize)

turnaround time: all the times added together, time from submission to completion (minimize)

Scheduling algorithms

Scheduling algorithms:

which one of ready processes to execute next

FCFS

1) First Come First Serve // not best algorithm

- non preemptive, Problem: a long jobs towards the beginning slows down everything
- (average wait time high)

SJF

2) shortest job first

- Preemptive / non preemptive // could be both
- optimal algorithm
- Problem: need to know jobs ahead of time (not practical)

PR

3) Priority scheduling

- Preemptive / non Preemptive // could be both
- CPU is allocated to the highest priority job
- Problem: low priority dont get to run (indefinite blocking) (starvation) (livelock)
- solution: Aging increases priority

RR

4) round-Robin

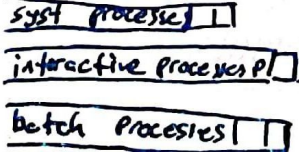
- preemptive
- every job gets a time quantum (slice) and back to ready que
- small timesice ⇒ more time context switching
- large timesice ⇒ reduces to FCFS

MQS

5) multi-level Que Scheduling

- Several ready queues
- one ready que per job Type (foreground, background, mem reqs)
- jobs are permanently assigned to ques
- Each que can use 1 through 4 scheduling (other types of scheduling)
- Fixed priority preemptive scheduling among queues

Fig 6.6



Scanned by CamScanner

MFQS
Best but hardest to implement

6) multilevel Feedback Queue scheduling
- generalization of 5 (MQS)
- Processes move between Queues

Fig 6.7 (example)

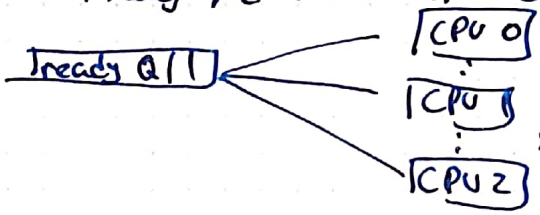
timeslice = 8 | |

timeslice = 16 | |

FCFS | | | |

7) Real time scheduling
- real time job - gets highest priority and its priority must not degrade
- as soon as it comes in it takes over and does not stop till done

8) multiple processor scheduling
- 1 ready que and multiple CPU's



- ready que is a shared object
- conflict in access
  - fix with semaphores

Sol 1 → - only one CPU runs scheduling
(only one cpu runs OS)
(Master-slave solution)
(interrupt CPU 0)

Sol 2: a copy of scheduler runs on all cpu
- ready q is accessed only in critical section

4 Processes w/ following CPU bursts

P1: 25 P2: 5 P3: 15 P4: 10

compute average wait time and average turn around time for FCFS, SJF, RR (TS = 4, 8)

First come First Serve

FCFS

P1 (25)	P2 (5)	P3 (15)	P4 (10)

0 25 30 45 55

average wait time $= \frac{0 + 25 + 30 + 45}{4} = 25$

average turn around time $= \frac{25 + 30 + 45 + 55}{4} = 38.75$

Smallest Job First

SJF

P2 (5)	P4 (10)	P3 (15)	P4 (25)

5 15 30 55

average wait time $= \frac{0 + 5 + 15 + 30}{4} = 12.5$

average turn around time $= \frac{5 + 15 + 30 + 55}{4} = 26.25$

Round robin

RR (4)

P1	P2	P3	P4	P1	P2	P3	P4	P1	P3	P4	P1	P3	P1	P1	P1

4 8 12 16 20 25 29 33 37 39 43 46 55

P2 done (P2=8) P4 done (P4=8) P3 done

wait time: P1 = 16 - 4 = 12
4 - 20 = 4
39 - 33 = 6
46 - 43 = 3
30

P2: 4 - 0 = 4
20 - 8 = 12
16

P3: 8 - 0 = 8
21 - 17 = 4
33 - 25 = 8
43 - 37 = 6
31

P4: =
29

average wait time = 26.5 $\frac{30 + 16 + 31 + 29}{4} = 26.5$

average turn around time $= \frac{21 + 39 + 46 + 55}{4} = 40.25$

RR(8) average wait time = 25.75
turn around time = 39.5

Scanned by CamScanner

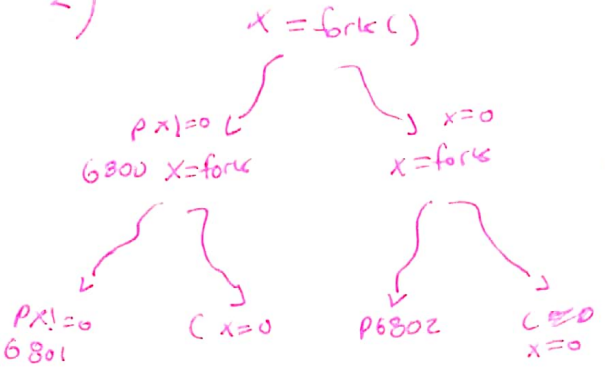
1) int in PCB

b) yes, ready Q is maintained based on priority: highest to lowest
no, every time the short term scheduler selects process with highest priority

c) it immediately runs

d) NO, aging increases priority

2)



order does not matter

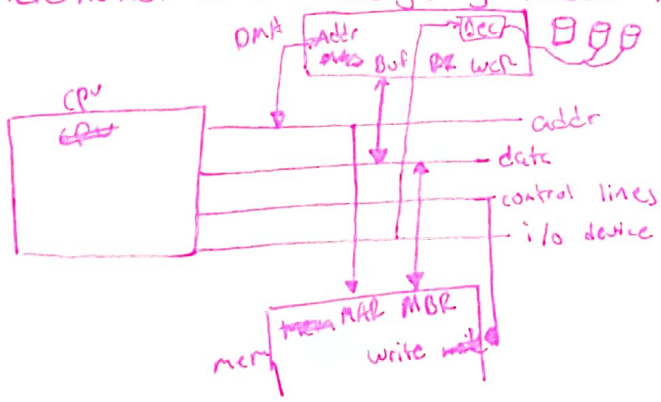
P 6800
C 0
P 6801
C 0
P 6802
C 0

4 Processes created

```
for (i=0; i<2; i++) {
    int x= fork();
    if (x!=0)
        cout<< "parent" << x;
    else
        cout << "child" << x;
}
```

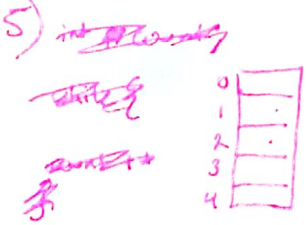
3) Dynamic linked modules can be added and/or replaced while OS is running
contrast this with layered approach, which is not possible
any module can call any other module -> more efficient, contrast this with microkernel which everything needs to go through communication facility

4)



1. CPU sets addr register = 4096 where data
2. DMA transfers one byte from device to buffer
MBR ← Buf, MAR ← AR, Mem write
Addr reg ++
wr --
while WCR≠0
DMA interrupts CPU

5)




```
int count = 0;
Producer:
  while (true){
    while (count == 5);
    ;
    buff [in] = item;
    in = (in+1) % BUF_SIZE;
    count ++;
}
```

```
consumer:
  while (true){
    while (count == 0);
    item = Buffer [out];
    out = (out +1) % BUF_SIZE;
    count--;
```

Scanned by CamScanner

Final

Dustin Bakiko
CSE460 Pg2
3/18/19

6.16 Priority: if you have a program that has to run
RR: allows for interaction
SJF: Fastest
LRU: need to use more hardware support

5 questions maybe 6 comprehensive
6.16 FCS
page replacement algorithm
semaphores - readers writers

what is hit ratio? % of time successful in page table
Falt ratio? % of time you get page falts

Scanned by CamScanner