Operating System

Done By:
Sarah Alkasasbeh

POWER UNIT

DATE

slides (4-8) . Lecture 1

. The main Component of the operating system :-
Kernel

. The operating system abstracts (simplifies) the Complexity of the hardware, and the RAM size ... etc. → (Abstracts)

. The operating system controls the hardware of the computer (ex:- CPU, memory, disk), it works to ensure a fair share of resources among different applications and users. (resources :- CPU, memory, disk)
↳ (Arbitrates)

. The hard disk is I/O device, so the main components to operate the Computer are the CPU and the memory.

. NIC (network interface card) is input and output device. "receives and sends packets"

. it is possible that there is a system that doesn't use an operating system, if there is only one application we want to execute on that hardware.

. The operating system is stored on the hard disk, or on external storage like flash.

slides (9-24) lecture 2

. The operating system deals with the controllers of the devices through the driver of the device, each component has different device driver.

. The alternative of the device driver is the ability of the operating system to deal with all hardwares or I/O devices. (not practical).

. Note:- controller :- hardware / Driver :- Piece of software

. every device controller has local buffer, to store enough data in it before it is used by the CPU, in order to avoid the speed mismatch between the I/O devices and the CPU.

. The interrupt temporary terminates the task of the CPU to do something else. dealing with interrupts is according to priority of eachone.

. hardware interrupt :- interrupt caused by wires in the CPU.

. software interrupt :- no wires in the CPU determine if there is interrupt. (ex:- I/O, operation).

. interrupt is external, but the exception is internal error.

DATE

- Two methods to handle interrupts :-

1- Blocking I/O → CPU becomes idle until the interrupts end.

2- Non-bloking I/O → The CPU completes exeauting the instructions. (it helps avoiding speed mismatch)

- BIOS (Basic input output system) → makes sure that there is not any problem in the main components, and it is the first one operate once turning on the device, then the bootstrap program runs.

- Bootstap stored in a permanent storage, it loads the operating system from the disk to the memory.

- first program runs.

- init → first service in the operating system, it initialices the system, operats memory managment service, operates CPU scheduling service .. eetc.

- Bootstrap program is typically stored in ROM or EPROM, generally known as firmware, (firmware ex :- bootstrap, ↓ BIOS)

any software that controls hardware.

DATE S M T W T F S

slides (25-37) lecture 3

- DMA (Direct memory access), works on transferring the data between I/O devices and main memory.
- before the DMA, transferring the data between I/O devices and memory was through the CPU, which affed the CPU time.

- Kernel : basic service in the operating system, controls all other services provided by the operating system.

- Hardware interrupt on kernel → by one of the I/O devices.

(OS elements) →

- one of the OS elements is the abstraction (like:- process, thread, socket...etc), and to deal with these abstractions we need mechanisms (like create, open, write... etc), so we need policies to avoid violation (like LRU, EDF).

ex:-(memory managment) →

- the OS allocate a place in the memory for each process (application) called memory page, it equals 4kB.
- LRU : swap between the "least recently used" page and other page the process needs. (LRU page from the memory to the disk).

• modes of operation :- 1- privileged mode (kernel mode) → means that the instructions are operating system's instruction (specifically related to the kernel).

2- unprivileged mode (user mode) → executed instructions are user application instructions.

• some instruction are not allowed to be executed on the user mode like halt instruction and timer instructions. (halt instruction → shut down the system)

• System call → when the user program tries to execute a privileged instruction, or to deal with an I/O device. → trap mode bit becomes zero → after finishing the system call and execute the instructions the trap bit becomes one, and the control returns to the user application.

• program is a passive entity (before execution)

• Process is an active entity (program in execution)

Mode bit = 0 → kernel mode }
Mode bit = 1 → user mode }

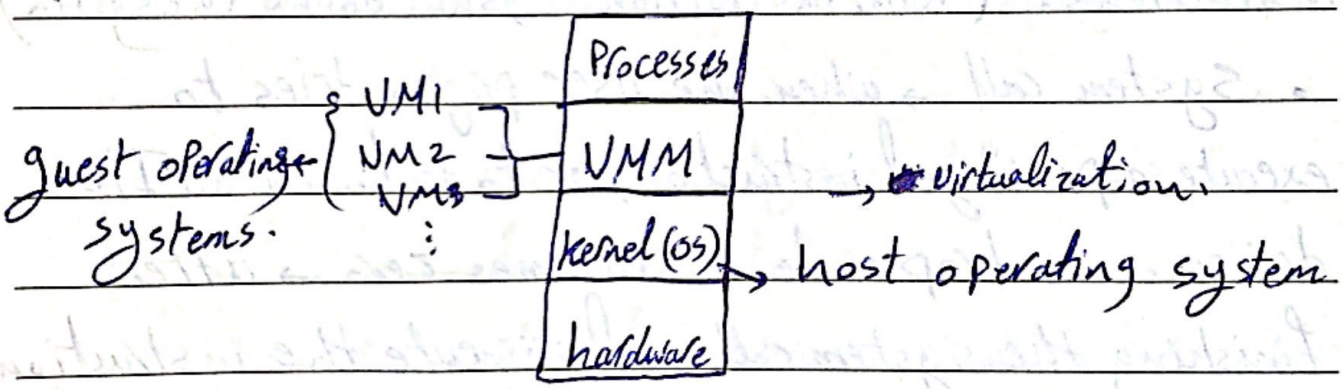
DATE S M T W T F S

slides (38-55) lecture 4

note
- interrupt number → interrupt vector → determine the interrupt service routine

* (difference between emulation, Virtualization)

- emulation :- Computer program's ability in an device to emulate the behavior of another program or device. (slow)
- Virtualization :- The process of creating a virtual instance of computer hardware platforms.



slide 40

- Local Area network (LAN) → building
- Wide Area network (WAN) → Countries
- Metropolitan Area network (MAN) → City
- Personal Area network (PAN) → one person

} Covered Areas.

↳ ex:- health care applications.

- Throughput :- number of tasks a system can process (finish) in a given amount of time.

DATE

. Fault Tolerance :- system's ability to continue operating uninterrupted despite the failure of one or more of its components. (the user wont notice the failure)

. Graceful degradation :- The ability of the system to maintain limited functionality (at reduced level) when portions of the system break down.

. System reliability :- related to the frequency of failure, when the system is reliable, the frequency of failure is minimal.

. Mesh topology :- Network setup where each component is interconnected with one another.
  . advantages → high performance
  . drowback → complexity, high cost

. Client server → The client nodes requests for services and server node responds with services.
. Peer to peer network (P2P) → each node can request for services and provide services. (needs more effort)

Cloud computing :- access different services through internet

DATE

Lecture 5

Open source operating system → operating system available in source-code format, it doesn't have to be free to use like the free operating system.

End of Chapter one.

slide(1-12) | Chapter 2

. System call → is a request from computer software to an operating system's kernel to perform a specific service.

. operating system services :- (user can notice) → provide useful functions

1- User interface (UI) → The part of an operating system that allows a user to enter and recieve information.

2- Program execution and identifying errors.

3- I/O operations (like dealing with files stored on the HD)

4- File-system manipulation → gives permission for operation on files.

5- Communications between different processes.

6- Error detection → The operating system keeps track of all the operations and processes and makes sure that they are running correctly without any errors, by using certain techniques.

DATE S M T W T F S

→ just to make the system runs efficiently (user can't notice)

operating system services :-

1- Resource allocation → the process of managing the different processes between hardware operational resources.

2- logging → to keep track with each user and their actions on the system, especially when there are multiple users.

3- protection and security

in System call → Control is transferred from user mode to the kernel mode

- System Programs → Programs used by the kernel & in user mode.

shells examples :- Csh, Sh, Bourne

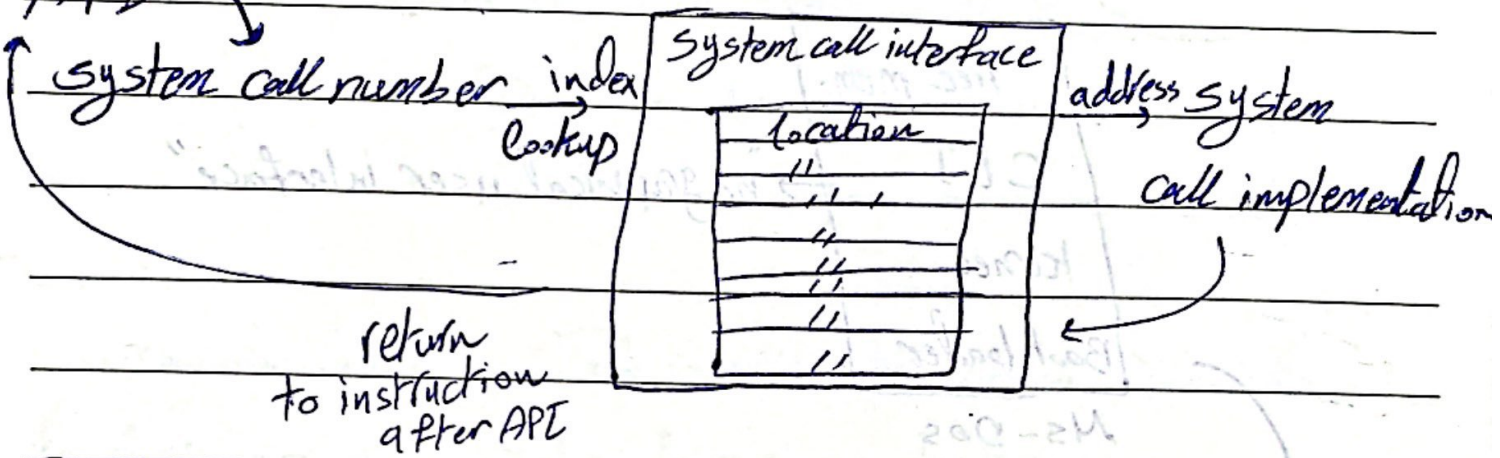
slides (13-24) lecture 6

- The transition from user mode to kernel mode occurs when the application requests the help of operating system or an interrupt or a system call occurs.
- if the privileged instructions are executed in user mode, it is illegal and a trap is generated.
- The mode bit is set to 1 in the user mode - it is changed from 1 to 0 when switching from user mode to kernel.

- To copy the contents of one file to another file we need a lot of system calls, at every step. Even if any error occurs, this requires a lot of system calls, so any command needs thousands of system calls.
(- any simple operation needs thousands of system calls)
- Programmers deal with API not the system call because the system calls is operating specific.
- each API has a specific system call that we can access through it. (ex:- read())

- The set of instructions to use or access a certain system call are in the kernel.

- each system call has a number that represents an index in a table of system calls in the system call interface.



- Usually the names of the system calls aren't identical to the names of the APIs, but there is a great similarity between them.

Note:- any access to I/O devices requires a system call. (ex:- open file stored in HD)

API contains → return value / function name / parameters

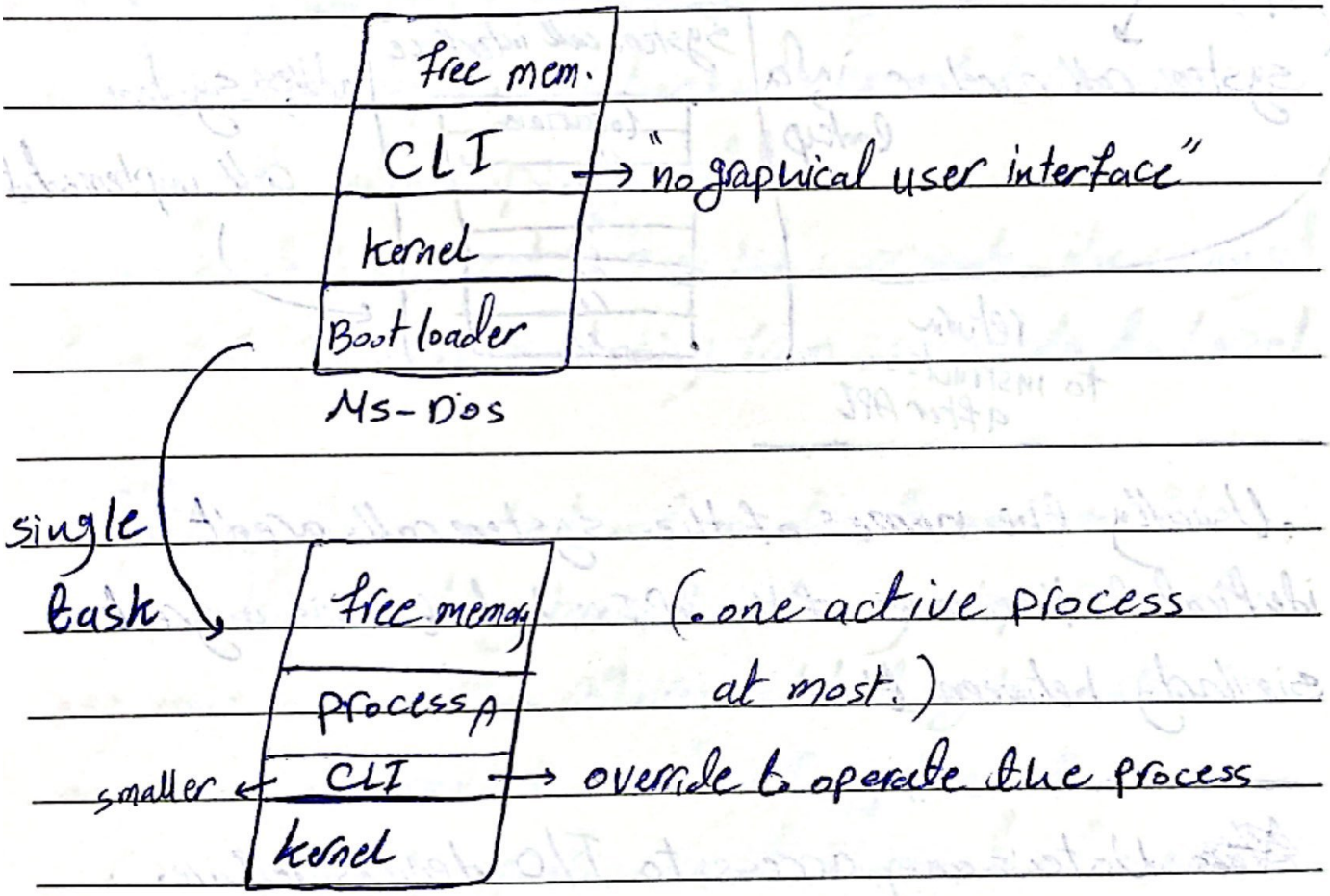
- Note:- libraries that are included maps the API with the system call (read (API) → open (S.C.))
- libraries have the implementation of each system call.

ATE

lectures (25-34) Lecture 7

- Single task ≡ No operating system

- MS-DOS :- single task operating system



- FreeBSD → multitasking operating system

- system calls → in the kernel

- system programs → services by the OS to provide a convenient environment for program development and execution → in the user mode

• most users' view of the operating system is defined by system programs and user application.

• Registry → manages resources and stores configuration settings for applications in the system (ex:- windows registry)

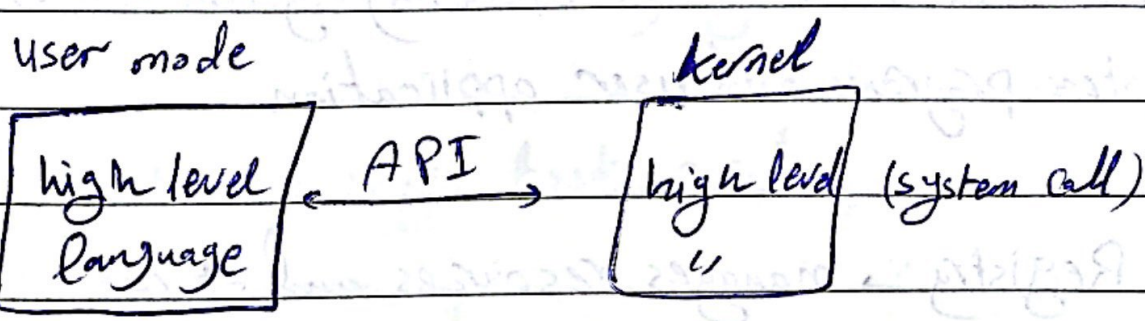
• remote log in :- The ability to access the data stored on a computer from a remote location.
↳ example of system services/programs
(application examples :- PuTTy, WinSCP
↓
transfer files

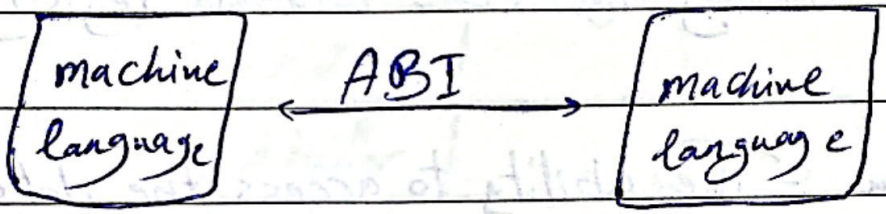
• The Role of linker /loader :-
linker → generate executable files
loader → load executable files to memory.
↳ system programs

• Apps compiled on one system usually not executable on other operating systems due to different system calls, file formats ... etc.

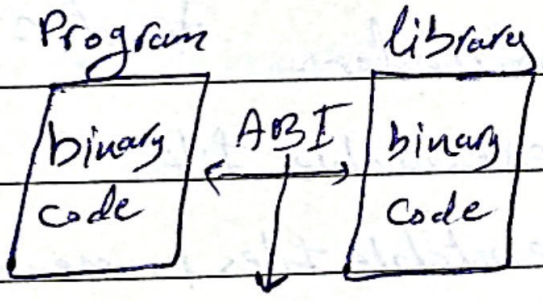
DATE

slide (34-42) lecture 8





linker :- system program, links two object files in one object file (library file and machine code file), gives executable file.



ex:- linker maps the abs value in the Program to the math library using ABI

Remember :- operating system elements :-

1- Abstraction
2- mechanism
3- Policy

Ex :- Policy :- LRU, LFU

mechanism :- Count the number of times a page is used

↳ the two policies may use the same mechanism

P_1	P_2	P_3	P_1	P_2	LRU → P_2
					LFU → P_3

- simple structure OS (like MS-DOS) → small kernel with few functionalities (limited), → fast boot
  - monolithic
- More complex OS (like UNIX) → very large kernel with a lot of functions, and number of services in the user mode much less than services in the kernel. slower boot and windows
  - ↳ system programs (very limited)

- layered operating system

use of

layers	modules (like linux)
↳ more complex is each layer contains certain functions	↳ The kernel executes a lot of functions that are divided into modules install OS → kernel with basic modules → other modules are installed by the user.

slide (43 - 54) Lecture 9

- microkernel → (like MACOSX) → Very small kernel but can be extended easily.
  in microkernel →
- Two processes can communicate through the kernel, using interprocess communication.
- Context switch is the process of storing the state of a process, so that it can be restored and resume execution at a later point. This allows multiple processes to share a single CPU. (switching between more than one process (ex:- user process and system call))
- push and pop and adjusting the PC counter reg.
- Context switching take time, so the higher the number of context switching, the lower the performance.

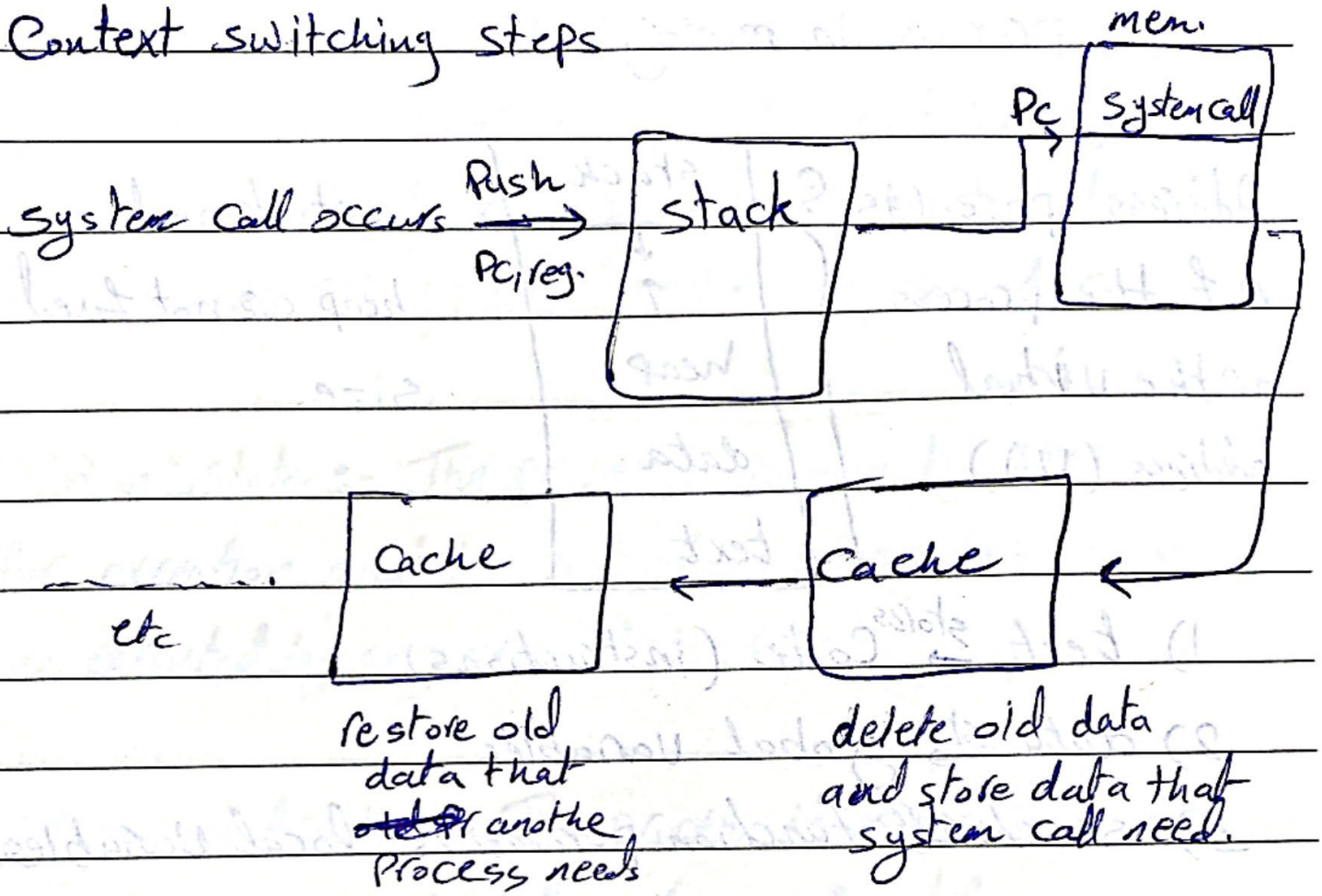
- difference between BIOS and uefi
  - ↳ The way they store information, configuration and setting
  - ↳ Updating BIOS firmware is a bit difficult.
- firmware → has direct control over low level hardware
  - ↳ ≡ BIOS.

- tcpdump is a packet analyzer that is launched from the command line. (alternative : wireshark)

DATE

S M T W T F S

• Context switching steps



cache hot → hit
cache cold → miss
} affect performance.

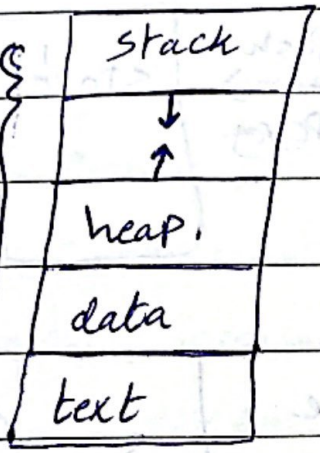
End of chapter 2

DATE S M T W T F S

Chapter 3 | slides (1-9) . lecture 10

• Process in memory

Address space (AS) of the process or the virtual address (VA)



• stack and heap are not fixed size. because the size depends on the application

1) text stores Code (instructions)
2) data stores global variables
3) stack stores function parameters, local variables, return addresses
4) heap → variables that are allocated at runtime. (ex:- by malloc function)

• global variables :- are not defined inside any function, and have a global scope.

• Process state

1) "New" state :- In this step, the process is about to be created but not yet created. (Command or double click)
2) "Ready" state :- The process is ready to run, after it has been approved by the operating system. (according to security and resources)

• Processes that are ready for execution by the CPU are maintained in a queue for ready processes (run/ready queue), each process has its own process ID (PID). unique integer number.

3) "Run" state :- The process is chosen by CPU for execution and the instructions within the process are executed by one of the CPU cores.

4) "terminated" state :- The process is killed, (finished) as well as PCB is deleted. (use "exit" system call)

• The short term scheduler or the dispatcher moves the process from ready state to running state.

• if the time allotted for the process has expired (run → ready) "interrupt".

• if the process needs any I/O operations during the running state (run → waiting queue) → (waiting queue → ready queue)

DATE

slides (10-29) Lecture 11

• process Control block (PCB) :- Data structure used by the operating system to store all the information about a process.

• Process scheduling allows OS to allocate a time interval of CPU execution for each process. Also, the CPU scheduler selects a process among the processes that are ready to execute and allocates CPU to one of them. → according to priority

• Child process :- process created by another process.

• Context switching time is affected by the PCB size.
PCB↓ time (ct-sw) ↓
as well as it is affected by the operating system itself and its complexity.

• The context switching is useful when :-
(2 ct-time < idle time)

Note :- Queues → tail points to the last process
head " " " process added first

• Operations on Processes :-

- Process creation → using certain system call.
- Process termination
- Scheduling
- Communication

Pid = fork(); → process creation, returns the new process id. &

(until terminates)

- if the parent process wait for the child process ↑ :-

• exec() system call → The previous executable file is replaced and new file is executed / ~~replace the old~~ all code (text) and data in the process is lost and replaced with the the executable of the new program.

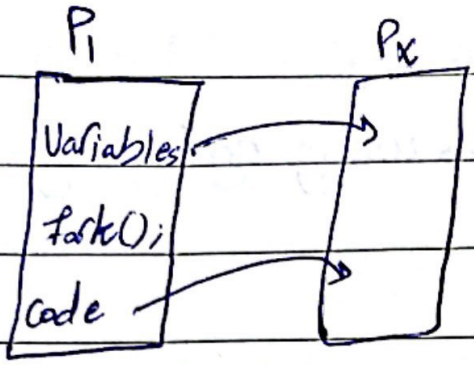
→ it makes the child process perform tasks different from the tasks of the parent process.

• wait() system call → Parent process waiting for the child to terminate.

• once the child process is terminated, it will return information to the "wait" system call (parent process) through the wait system call ~~&~~ using exit() system call.

DATE

slides (23-27) lecture 12



Px takes the same code after fork in P1 and only variables before fork.

- The parent process (id = id returned from fork();)
- child process (id = 0)

Ex:-

```
Cout << "Hello OS" << endl;
type ← Pid_t Pid = fork();
Cout << "Bye, see you later!" << endl;
```

output:-

```
Hello OS
Bye, see you later!   → parent
Bye, see you later!   → child
```

Note: exec() → take 2 parameters, Pid of the process which perform the task, and the new executable file.

- if fork() returns a negative value → operation failed.

Ex:- Cout << "Hello OS" << endl;

Pid_t Pid1 = fork();
Pid_t Pid2 = fork();
} → 4 processes, 2 parents, 2 children

Cout << "Bye, see you later!" << endl;

output:-

output after fork = $2^{\#\text{ of forks}}$

Hello OS
Bye, see you later!

3 forks → 8 times
→ 4 times.

- A call to wait() blocks the calling process (Parent) until its child process exits. After child process terminates, Parent continues its execution.

- Once the Parent process terminates, all of its children Processes should terminate too. (best practice)
else → if the children Processes don't terminate with the Parent process they become called orphan processes. (they keep the resources).

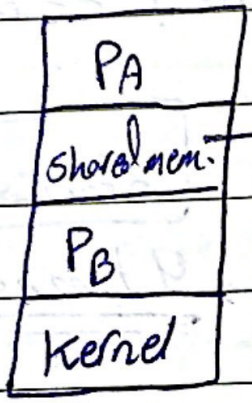
- Multiprocess Architecture → better performance than Single process Architecture.
→ in Browsers → each tap is separate process.
Sandbox → impove security by isolation and restrictions

DATE

slide (28-38) . lecture 13

. Communications Models :-

a) shared memory



→ accessible by PA and PB, both of them can read from it, or write on it.

↳ it is located in the address space (AS) of the process that requested communication.

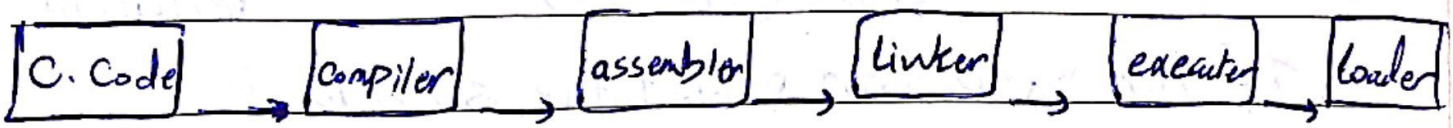
. The function of the kernel here is to allow one process to access the address space of the other process which has the shared memory.

b) message passing

. Communication in this model is through the kernel.

. The kernel here is involved in the communication process.

DATE



. Compiler/assembler → Compiler is producer, and assembler is Consumer.

. assembler/linker → assembler is a produer, and linker is Consumer.

etc.

. Two Variations (producer-consumer problem)

1- unbounded buffer : no practical limit on the size of the shared memory.

2- bounded buffer : shared memory with fixed size.

• Synchronization problem in the shared memory model:-
- two processes need to access the same area (shared area) at the same time (without any synchronization)

~~• buffer size / shared memory size~~ → circular buffer → bounded buffer

in / out values in the buffer (shared memory)

else (should be in range (0 to size-1)

#. Counter variable = # of items in the buffer, (valid data)

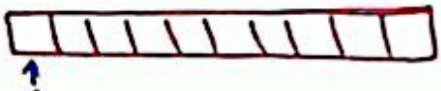
↳ 3 steps instruction.

شايفينه التنين الـ buffer / in/out ← Consumer وال Producer.

Bounded-Buffer – Shared-Memory Solution

```
#define BUFFER_SIZE 10
typedef struct {
    . . .
} item;
```

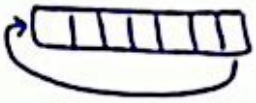
شبه الـ class بالجافا مش مهم التفاصيل.



بالبداية in و out ← حيأشروا كنفس المكان يعني فاضي الـ buffer

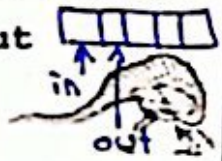
```
item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

bounded array (Circular Array)



- The shared buffer is implemented as a circular array with two logical pointers: ***in*** and ***out***
  - ***in*** points to the next free position in the buffer → بالبداية بكون 0
  - ***out*** points to the first full position in the buffer.
  - The buffer is empty when `in == out`
  - The buffer is full when `((in + 1) % BUFFER SIZE) == out`
- Solution is correct, but can only use `BUFFER_SIZE-1` elements

بكون الـ buffer مليان لو الـ in كان قبل الـ out بوحدة م



عملتها عشان

Circular تخليه بتعالج حالة لو كان الـ in عآخر الـ Pointer فالـ out حيكون أول الـ array لإنه Circular.

ما بقدر أستخدم كل الـ Size تاع الـ buffer لأنه بضل في وحدة empty بتاعة in.

Producer Process – Shared Memory

local variable ←

```
item next_produced;

while (true) {
    /* produce an item in next produced */
    while (((in + 1) % BUFFER_SIZE) == out)
        ; /* do nothing */
    buffer[in] = next_produced;
    in = (in + 1) % BUFFER_SIZE;
}
```

infinite loop ← while (true) {

معناها الbuffer full ما بقدر أحط لحد ما الconsumed يفضي و يحرك الout ويصير عندي مساحة أحط.

* اللي بوقف الـ producer إنه يكون الـ buffer (full) ، واللي بوقف الـ consumer يكون الـ buffer (empty)

Consumer Process – Shared Memory

```
item next_consumed;

while (true) {
        while (in == out)
                ; /* do nothing */
        next_consumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;

        /* consume the item in next consumed */
}
```

* بحالة الـ Full دايمًا في element مش معبى ، ما بقدر أمشي الـ in خطوة ويصير بيساوي الـ out .

* هاي الحالة ما فيها Race conditions لأنه الـ Race ما بصير بالـ local variables بصير بالـ Shared var .

← الـ buffer والـ in والـ out هم الـ Shared ،

- in=out بس لما يكون الـ buffer ← empty فما في overwrite عالـ buffer

- الـ Producer بعمل update للـ in والـ Consumer للـ out فما في (Race conditions) بينهم

Producer

```
while (true) {
        /* produce an item in next produced */

        while (counter == BUFFER_SIZE)
                ; /* do nothing */
        buffer[in] = next_produced;
        in = (in + 1) % BUFFER_SIZE;
        counter++;
}
```

حالة الـ Full → while (counter == BUFFER_SIZE)

حرنا نعتمد على counter لتحديد الـ buffer لا فاضي أو مليان مشان حال in والـ out.

Consumer

```
while (true) {
        while (counter == 0)
                ; /* do nothing */
        next_consumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        counter--;
        /* consume the item in next consumed */
}
```

Empty →

لحد ما يصير في →

← item جوا buffer الـ

* هيك حلينا مشكلة إنه فيه element بضل فاضي بحالة الـ Full
بس صار فيه Race condition لإنه الثنين
بعملوا عالـ counter

Race Condition

- `counter++` could be implemented as

load ← `register1 = counter`
increment ← `register1 = register1 + 1`
Store ← `counter = register1`

- `counter--` could be implemented as

load ← `register2 = counter`
decrement ← `register2 = register2 - 1`
Store ← `counter = register2`

- Consider this execution interleaving with "count = 5" initially:

S0: producer execute `register1 = counter`	{register1 = 5}
S1: producer execute `register1 = register1 + 1`	{register1 = 6}
S2: consumer execute `register2 = counter`	{register2 = 5}
S3: consumer execute `register2 = register2 - 1`	{register2 = 4}
S4: producer execute `counter = register1`	{counter = 6 }
S5: consumer execute `counter = register2`	{counter = 4}

الثاني قرأ الـ counter قبل ما تتعدل
وهيك صار incorrect لأنه كان لازم تضل 5 .

slides (39 - 47) lecture 14

- If the "in" pointer points to the same location as the "out" pointer, we assume that the buffer is empty, or counter = 0.

- Atomic operation :- operation that cannot be interrupted while being executed

  ex :- register1 = counter

- no race condition in atomic operations

(counter++) is not atomic because its 3 steps instruction

- Question - why was there no race condition in the solution (where at most N-1) ?

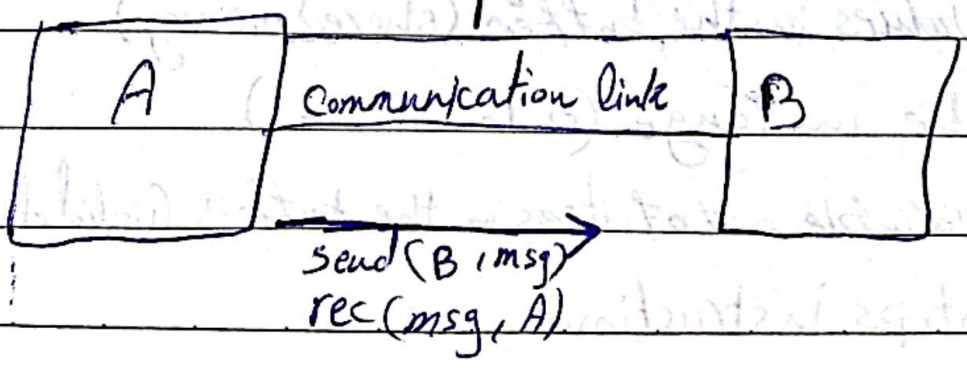
  : because there aren't any shared variables and they aren't modify the same variables.

- message passing model :-

  if the message size is fixed → easier in system implementation.

  - if the message size is'nt fixed → easier for the programmer.

: dired communication



DATE

slides (48-56) Lecture 15

- Buffering
  - we have a link between 2 processes, this communication link uses a buffer which is a queue of messages, and attached to the link. It is used to avoid losing messages in case of speed mismatch
  - This buffer could be :-

1- zero capacity → if the sender send a message, the reciver should recieve it directly, otherwise the message will be lost. (no buffer)

- The sender most remain blocked until the recipient recives the message.
- If the receiver uses the (receive system call) before the sender send a message, the receiver will receive a NULL message, so it must remain blocked too
- in zero capacity → send, receive are blocking system calls.

2- bounded capacity → (real life scenario)

3- Unbounded capacity

- factors taken into consinderation to determine the size of the buffer :-
  1- Communication rate
  2- speed mismatch between sender and receiver
     - if producing speed is faster than the speed of consumption → large buffer
     - if producing speed is slower than the speed of consumption → could be zero capacity buffer.

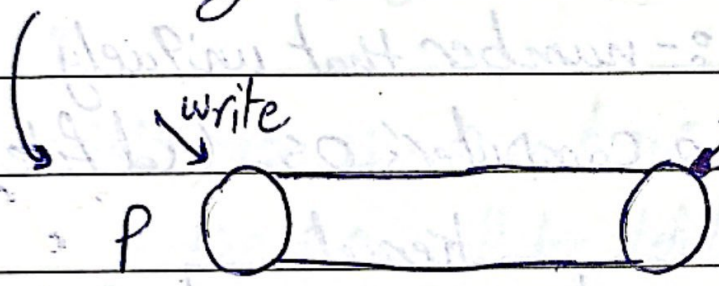
- RPC (remote procedure calls) occurs when there is communication between two processes, each one is on different system, in (RPC) the socket is created. (socket is the endpoint of communication link) (socket is a combination of an IP address and a port number).
  - Port → represents the type of the service
    well known services (0 - 1024)
    ex:- http in port 80 (web server)

DATE

S M T W T F S

Pipe:- implementation of the communication link between two processes which communicate through message passing.

- Ordinary Pipes → unidirectional



- parent - child relationship.

- Named pipes → unidirectional but not Parent - child relation ship

Note:- half - duplex → send or receive at a specific moment.

Full - duplex → Processes send and receive messages at the same time.

DATE

slides (55 - additional) lecture 16

• ordinary pipe (unnamed) pipe lasts only as long as the process. → created by the process

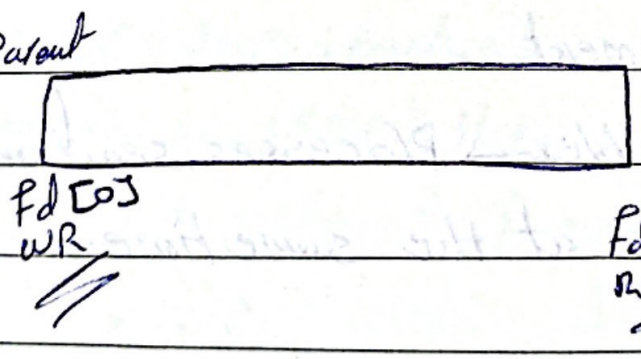
• named pipes can last as long as the system (kernel) is up, beyond the life of the process. → created by the kernel.

• fd (file descriptor) :- number that uniquely identifies an open file in a computer's OS. (id file)

• std in :- standard input
↳ ex :- keyboard

• std out :- default/standard out
↳ ex :- monitor

• std error :- where a process can write error message.



int myFD[2]; → create 2 fd (for write and read)
Pipe (myFD); → API to create a pipe
Pid_t x = fork(); → creat a child (ordinary pipe)

if (x>0) { → Process ID >0 → Parent
write (myFD[1], "hello child", 30); → message size
}
else { → child
int n= read (myFD[0], char msg); → returns # of read charac.
}

Note:- once the parent process is terminated, the pipe will be deleted.

Note :- if we use (write (1, "hello child", 30);) the message will appear ↵ on the monitor because the FD of the (std out) is 1

- Bidirectional pipes → only using 2 pipes.

Chapter 4 slides (1-6)

Process :- Program in execution, and the execution style is "sequential" → (there is no Parallisim in the Process). if one instruction is blocked, then all the other instructions will be blocked too.

↳ Therefore, the main process is divided into sub processes

DATE

S M T W T F S

. if one of the sub processes is blocked, other sub processes won't be affected.

. each sub process is called "Thread"

• multithreaded process → Parallel execution.

. The code and Data segment is shared between all threads that are related to the same process, as well as files.

• each Thread has a unique ID, PC, stack and registers

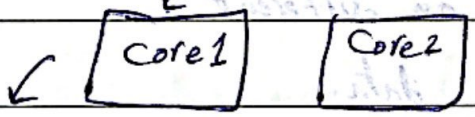
. It is possible to split data between threads so that each thread performs the same instructions on a specific set of data, or to split the instructions so that each thread performs a different task.

DATE

slides (7-9) type of context switching / lecture 17

. Thread switching → switching from one thread to another thread in the same process.
Thread switching is very efficient and cheaper than process switching because it involves switching out the PC, registers and stack pointers. (lower overhead)

. MTP (T_1, T_2)



. any blocking or I/O operation on Core 1 (T_1) won't affect the execution on Core 2.

→ multithreaded process takes advantage of multicore arch.

→ It is preferable to use a number of Threads equal to the number of cores.

→ multithreaded processes take advantage of single core arch. too.

. Parallel execution → multi different processes (task) on the same time. → multicore system. (2 cores → 2 threads or more)

. Concurrent execution → only one thread (task) is executed at a time. → in single core system.

slides (10 - 20) lecture 18

- Data parallelism :- Distribute data to more than 1 thread while performing the same operations on it.

ex:- summation on elements in an array.

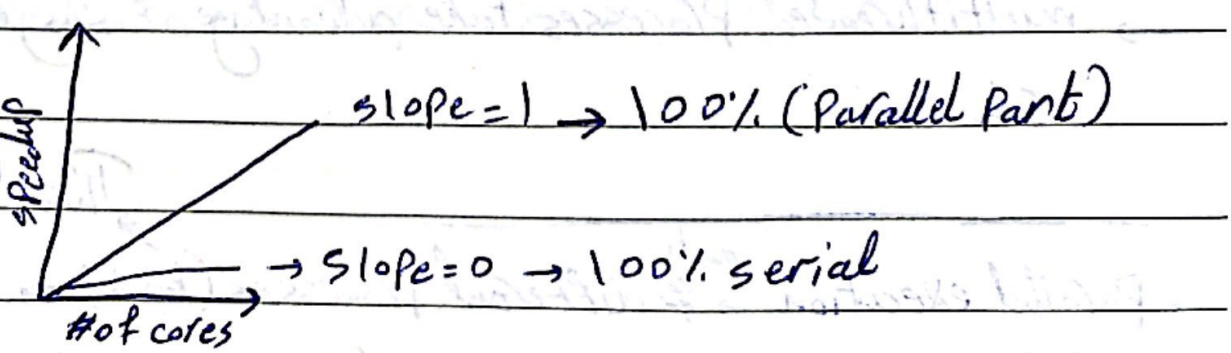
- Task parallelism :- distributing thread across cors, each thread performing unique operation.

on same data | on different data.

ex:- Performing different operations on an array.

Note :- Parallel part = (1 - serial part).

- Inverse proportion between the serial portion and the speedup that can be obtained.



- We cannot have a line slope of more than 1.

. if the slope = 0.5, find the percentage of serial and parallel portion?

. Kernel is multithreaded, to serve the requests that are from the user mode.

. Threads in user mode are managed by library in the user level.
. The threads that are in the kernel mode, are managed by the kernel.

. Multithreading models :- relationships between user threads and kernel threads.

1. many-to-one → no parallel execution. (no limitations on # of user threads)
2. one-to-one → kernel threads = user threads
   - There is parallel execution.
   - There is a limitation on the # of user threads. to avoid overhead.
3. many-to-many → UT ≥ KT / no limitations / parallel execution

Amdahl's Law

18

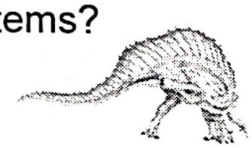
- Identifies performance gains from adding additional cores to an application that has both serial and parallel components
- *S* is serial portion
- *N* processing cores

$$speedup \leq \frac{1}{S + \frac{(1-S)}{N}}$$

- That is, if application is 75% parallel / 25% serial, moving from 1 to 2 cores results in speedup of 1.6 times
- As *N* approaches infinity, speedup approaches 1 / *S*

  Serial portion of an application has disproportionate effect on performance gained by adding additional cores

- But does the law take into account contemporary multicore systems?



slides (21 - 33) lecture 19

- any Variable in the Data segment (global Variable), are shared between all threads in the Process.

- pthread_join (API used to prevent the main thread from terminating the process before the created threads have finished their tasks.). without this API the output won't be correct.

- In windows, when creating a thread we need to define HANDLE first.
- same other steps are in all threading libraries.

↑ level of parallelism ↑ creating, managing threads

↓

to avoid this → implicit threading

Thread pools → pool of threads is created by the compiler, where they await work. (its size is determined by compiler)
↳ faster than thread creation while execution. serving requests is faster.

Pthreads Example

```
#include <pthread.h>
#include <stdio.h>

#include <stdlib.h>

int sum; /* this data is shared by the thread(s) */
void *runner(void *param); /* threads call this function */

int main(int argc, char *argv[])
{
   pthread_t tid; /* the thread identifier */
   pthread_attr_t attr; /* set of thread attributes */

   /* set the default attributes of the thread */
   pthread_attr_init(&attr);
   /* create the thread */
   pthread_create(&tid, &attr, runner, argv[1]);
   /* wait for the thread to exit */
   pthread_join(tid,NULL);

   printf("sum = %d\n",sum);
}
```

global variable shared between all threads

default attribute

function to execute

Parameters

waiting for the created thread to terminate first before the main thread.

Pthreads Example (Cont.)

```
/* The thread will execute in this function */
void *runner(void *param)
{
   int i, upper = atoi(param);
   sum = 0;

   for (i = 1; i <= upper; i++)
      sum += i;

   pthread_exit(0);
}
```

→ sum numbers from 1 up to Param.

↳ termination, return to main function

Pthreads Code for Joining 10 Threads

```
#define NUM_THREADS 10

/* an array of threads to be joined upon */
pthread_t workers[NUM_THREADS];

for (int i = 0; i < NUM_THREADS; i++)
   pthread_join(workers[i], NULL);
```

tid ⇒ workers[i]

↳ array with size 10, each element is pthread type

slides (34 - 37) lecture 20

- Performance in multithreaded programs depends on the hardware (# of cores), so we need to determine the best # of threads.
In "Implicit Threading", This task which consists of specifying the number of threads, and task of each thread becomes to the compiler istead of the programmer.

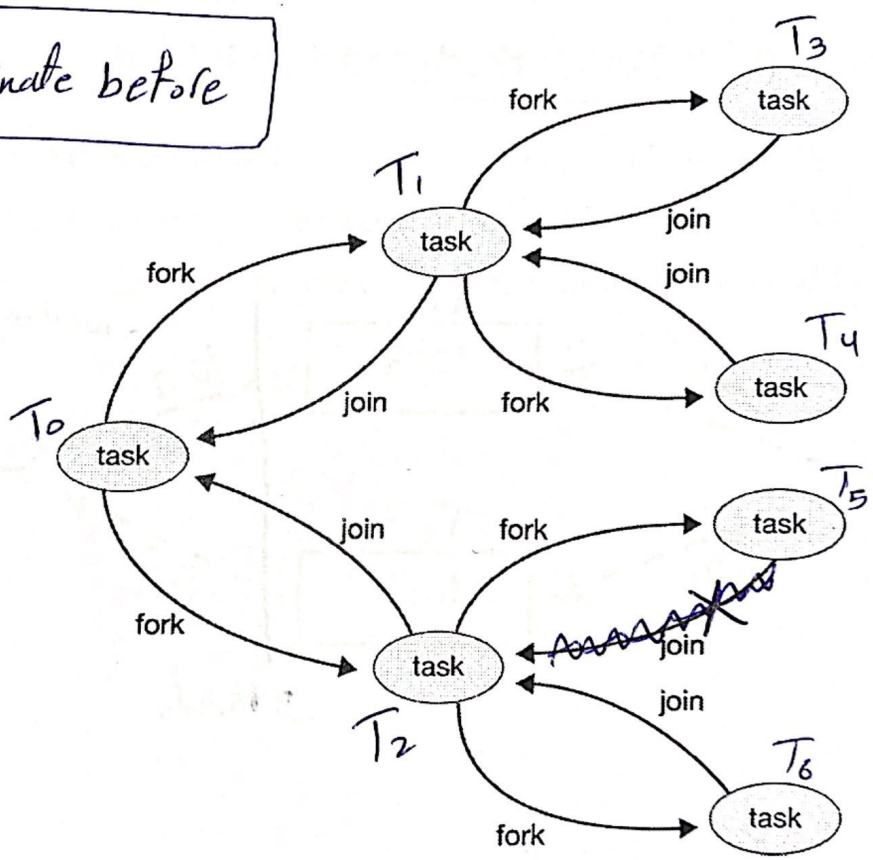
- in Implicit threading the compiler create the threads, and distribute the tasks between them. (5)

- Note :- In linux (Thread ≡ Task)

- The Parent thread cannot terminate its work before its child thread has terminated. Unless the child thread doesn't (Join).

- In openMP we have to determine the parallel region.
- The number of times the instructions within the Parallel region are executed is equal to the # of Threads / cores → In openMP.

To can terminate before T_5



OpenMP

- Set of compiler directives and an API for C, C++, FORTRAN
- Provides support for parallel programming in shared-memory environments
- Identifies **parallel regions** – blocks of code that can run in parallel

```
#include <omp.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
  /* sequential code */

  #pragma omp parallel
  {
    printf("I am a parallel region.");
  }

  /* sequential code */

  return 0;
}
```

`#pragma omp parallel`

Create as many threads as there are cores

Threads = Cores

if Dual core this instruction will be executed two times (T_1, T_2)

quad → 4 times

slides (~~Feb~~ 43-47) lecture 21

- overloading :- multible functions with the same name, but different implementations.

~~a different~~

① Pa

```
fork();
exec();
```

2 different forks in UNIX :-

→ ~~only~~ fork only duplicates the calling ~~th~~thread. ex:- T_1, T_2, T_3

↓

- duplicates calling thread only

↳ T_4 → new code

② Pb

```
fork();
```

→ fork duplicates all threads.

ex:- T_1, T_2, T_3

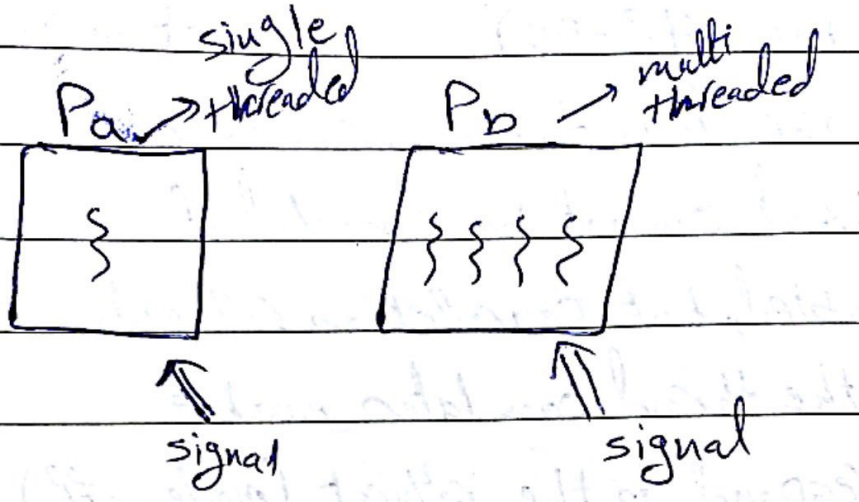
duplicates all ~~th~~threads.

↳ T_4, T_5, T_6

- Synchronous signal :- signal generated by the process itself (ex:- division by zero, access illegal memory location) → internal source

- ~~a~~asynchronous signal :- from external source. (ex:- I/O operations)

- There is default handler ~~wh~~in the kernel executed in case the user doesn't override the handler.



. In single threaded process, the thread recieve the signal, and handle it.

. In multi. threaded process :-

1) Deliver the signal to the thread whose task caused it. (ex:- division by zero)
2) Deliver the signal to every thread in the process.
(ex:- Termination process signal)
3) Deliver to a certain thread.
(ex:- "kill" system call)
4) Deliver all signals to a certain thread.

. Asynchronous Cancellation → The target thread should terminate its task immediately without completing any extra work after cancellation. → causes some errors like cancel before modifying certain variable.

DATE

slides (48-55) lecture 22

. Pthreads allows threads to disable or enable. cancellation. a thread cannot be canceled if cancellation is disabled, but cancellation requests remain pending, so the thread can later enable cancellation and respond to the request (mode: off)

. Asynchronous cancellation :- one thread terminates immediately the target thread.

. Deffered cancellation :- one thread indicates that a target thread is to be canceled, but cancellation occurs only after the target thread has checked a flag to determine whether or not it should be canceled.

. after cancellation, the "cleanup handler" invoked, to free the resources that a thread may hold at the time it terminates.

Thread-local storage :- Allows a thread to store data that is visible to all the function invocation within that thread, but it is invisible to other threads.

DATE

. The LWP appears to be a virtual processor on which the application can schedule a user thread to run, to the user thredd library.

↑ LWP ↑ overhead → more read and write operations

. How many LWPs to create? according to the upcalls and upcalls ~~hand~~ handler.

Note :-

. CPU-bound process :- process that spends the majority of its time using the CPU (ex:- calculations)

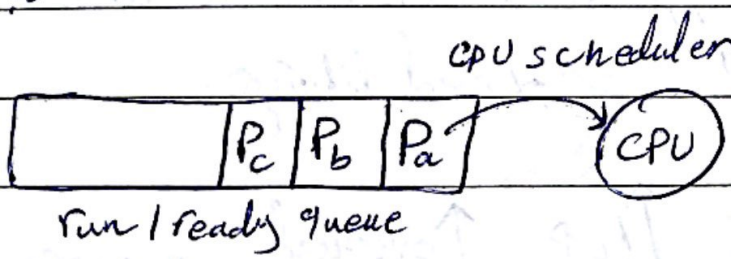
. I/O-bound Process :- process that spends the majority of its time doing I/O operations. It needs more kernel level threads than the CPU-bound process due to a lot of blocking system calls.

"End of chapter 4"

DATE M T W T F S

Chapter 5 slides (1-10) lecture 23

remember:-

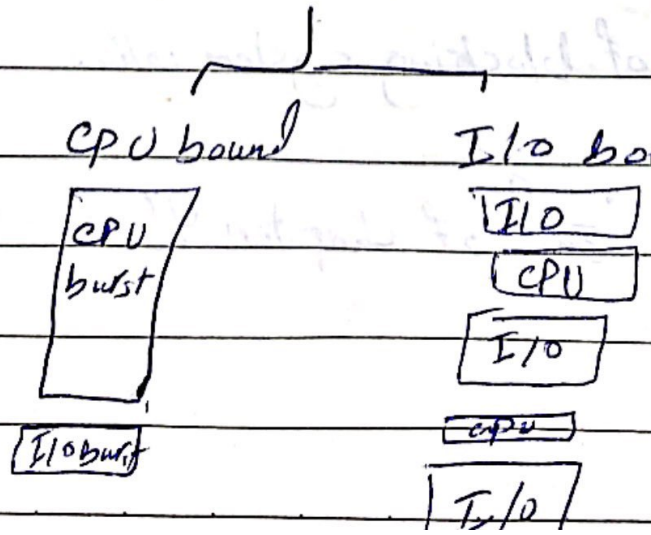


. if Pa needs any I/O operation, the CPU will start working on Pb

. Profiler :- software that examine, analyze, and create useful summaries of the execution of a process.

. execution of any process is CPU burst and I/O burst. Their distribution affects CPU utilization.

. # of CPU, I/O bursts and their length depends on the nature of the process



• switching from running to waiting state → when the process needs any I/O device, then the scheduler will determine which process from the ready queue should allocates the CPU.

• switching from running to ready state → when the process time expires, " " " "

• switching from waiting to ready → when the process is finished with the input and output device.

• resources utilization in preemptive scheduling is better, but can result in race conditions.

• scheduler works in kernel mode

• Dispatcher workes in kernel mode then switches to user mode.

Dispatch latency > context sw. time.

---

scheduling Criteria => scheduling goals or objectives

Turnaround time is the Completion time (CPU time / I/O time / waiting time)

slides (11 - 18) lecture 24

- optimization in any system design is trade off, so we have to work on the balance of any improvment or optimization, so that we don't affect the rest of the tasks in a negative way.

scheduling algorithms:-

1) FCFS → in the order they reach the ready queue.

(slide 12) → CPU utilization = $\frac{30}{30}$ = 100% (30 → CPU is busy; 30 → total time)

→ average Turnaround time (completion time) = $\frac{24+27+30}{3}$

→ Throughput = $\frac{\#\text{ of completed tasks}}{\text{time}}$

$= \frac{3}{30} = 0.1$ P/ms

2) SJF → Associate with each process the length of its next CPU burst → better average waiting time but we need to determine the next CPU burst length

S M T W T F S

Slides 20-24 lecture 25

- Scheduling Algorithms → decide which and when the process should leave the CPU.

ex:- non-preemtive SJF → the process won't leave the CPU, until it finishes its burst.

- preemtive SJF → The process will be ~~preempted~~ preempted if there is another process with CPU burst < remaining time.

ex slide 20

FCFS
non-preemtive

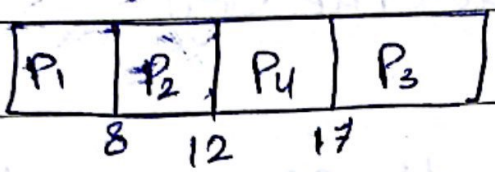
P1	P2	P3	P4
8	12	21	26

- wait time (P_2) = 8 - 1 = 7 (1 = arrival time)
- wait time (P_3) = 12 - 2 = 10
- wait time (P_4) = 21 - 3 = 18
- average turnaround time = wait time + Burst time
  = $\frac{8+11+19+23}{4}$ (4 = # of Processes)

- if there are more than 1 wait times they are all included in turnaround time.

ex. slide 20

Algorithm → SJF (non-preemtive)



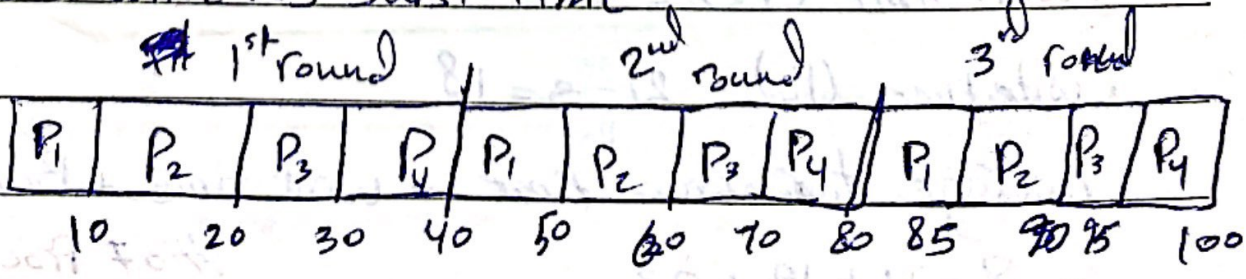
- wait time (P_1) = 0
- wait time (P_2) = 7
- wait time (P_3) = 15
- wait time (P_4) = 9

- FCFS → affects interative processes if they're arrive to the CPU late, after long processes.
- SJF → improves performance

Round Robin ex:-

$q_t = 10$

4 Process with 25 burst time



Max response time → $n-1 * q = 3 * 10 = 30$

DATE

- $q \downarrow$ ↑ context switching time ↑ overhead
- $q \uparrow$ → FCFS

- best value for (q) → { q > 80% of CPU burst time

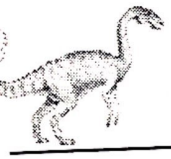
ex:-

P_1	P_2	P_3	P_4	P_5
2	3	4	5	6

$q > 5$

- Response time in RR is better than the response time is SJF.
- Average TA time in RR is greater than the average TA time in SJF. (SJF is better in TA)

- $q >$ context switching time.

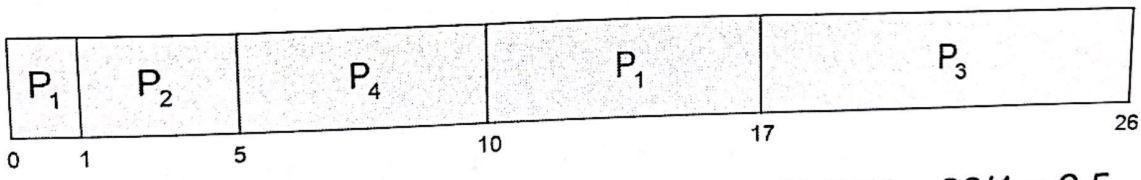


Example of Shortest-remaining-time-first

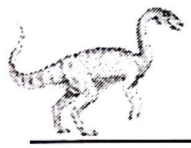
- Now we add the concepts of varying arrival times and preemption to the analysis

ProcessAarri *Arrival* TimeT		Burst Time	WT
P_1	0	8	9
P_2	1	4	0
P_3	2	9	15
P_4	3	5	2

- *Preemptive* SJF Gantt Chart



- Average waiting time = [(10-1)+(1-1)+(17-2)+(5-3)]/4 = 26/4 = 6.5



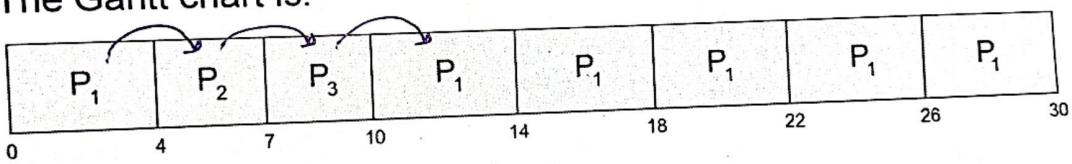
Example of RR with Time Quantum = 4

FCFS

Process	Burst Time	WT	TA
P_1	24	10	34
P_2	3	4	7
P_3	3	7	10

3 times → Context switching

- The Gantt chart is:



- Typically, higher average turnaround than SJF, but better ***response***
- q should be large compared to context switch time
  - q usually 10 milliseconds to 100 milliseconds,
  - Context switch < 10 microseconds

- response time $P_1 = 0$
- " " $P_2 = 4$
- " " $P_3 = 7$

DATE S M T W T F S

slides (25-34) lecture 26

priority scheduling

- each process has an integer number which represents its priority (smallest integer → highest Priority)
- highest priority → lower burst time
  ↳ so its type of SJF.

ex. slide 25

- wait time (P_2) = 0 (0-0)
- wait time (P_5) = 1 (1-0)
- wait time (P_1) = 6 (6-0)

:

- response time (P_4) = 18

in Example slide 26 :-

10 non-Preemtive

P_1	P_2	P_5	P_3	P_4

arrival time
0
10
4
2
3

• if there is more than 1 process with the same priority level, we can use priority scheduling with other scheduling algorithms like Round-Robin, FCFS, or SJF.

Run-to-completion is similar to non-Preemtive

• Searching in ready queue takes O(n), where "n" is the number of processes in the queue.

↓

So → use multilevel queue.

• In multilevel queue, we use round robin at the same queue, and priority scheduling between different queues.

Thread scheduling

• Scheduling in user level threads is done by Threading libraries. → many to many model / many-to-one
• Scheduling in kernel level threads is done by cpu scheduler.
• in one-to-one model → no need to schedule in the user level threads.

DATE

slides (34-44) lecture 27

. In windows and linux the scope is → SCS, because they are one-to-one model

. Multicore CPUs are faster, and less power consumption

. Hardware threading (2 register sets or more on the same core " 2 threads or more on the same core ") → multithreaded cores.

. Homogeneous multiprocessing → Identical cores

multicore → multithreaded cores → NUMA systems

. Heterogeneous multiprocessing → each core has different tasks.

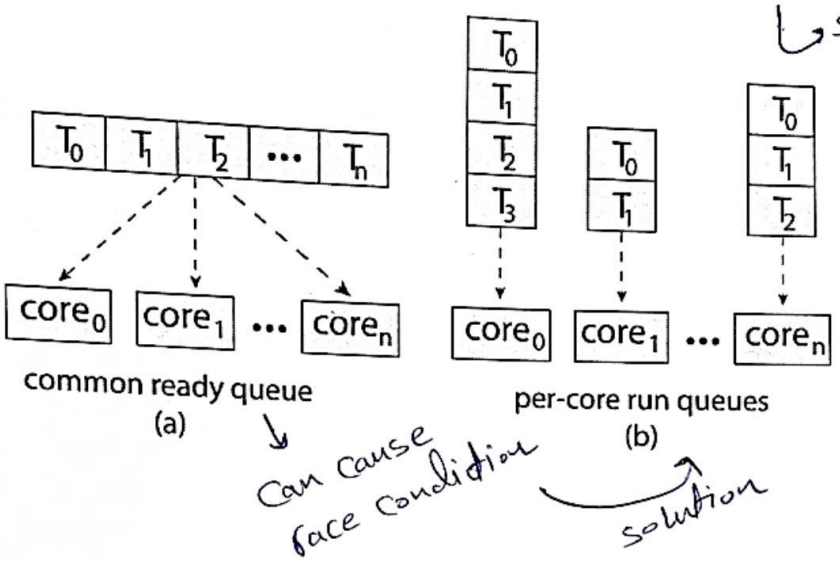
. Memory stall frequency depends on the code itself, size of the cache, and the locality.

. Memory stall time depends on the speed difference between CPU and RAM, and the RAM size.

Multiple-Processor Scheduling

- Symmetric multiprocessing (SMP) is where each processor is self scheduling.
- All threads may be in a common ready queue (a)
- Each processor may have its own private queue of threads (b)

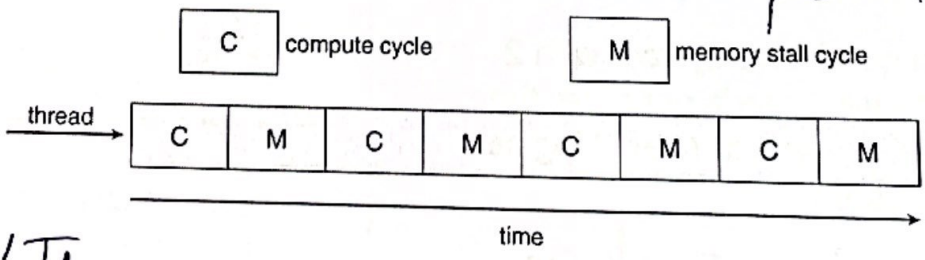
↳ should use load balance.



Can cause race condition → solution

Multicore Processors

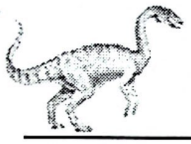
- Recent trend to place multiple processor cores on same physical chip
- Faster and consumes less power
- Multiple threads per core also growing HW threading
  - Takes advantage of memory stall to make progress on another thread while memory retrieve happens
- Figure



→ read/write from/to memory

T_0 / T_1

↳ one of them to memory stall, and the other one continues its work, to avoid overhead.



Multithreaded Multicore System

- **Chip-multithreading** (CMT) assigns each core multiple hardware threads. (Intel refers to this as **hyperthreading**.)

if the core supports 4 HW threads, then there are 4 Program counters.

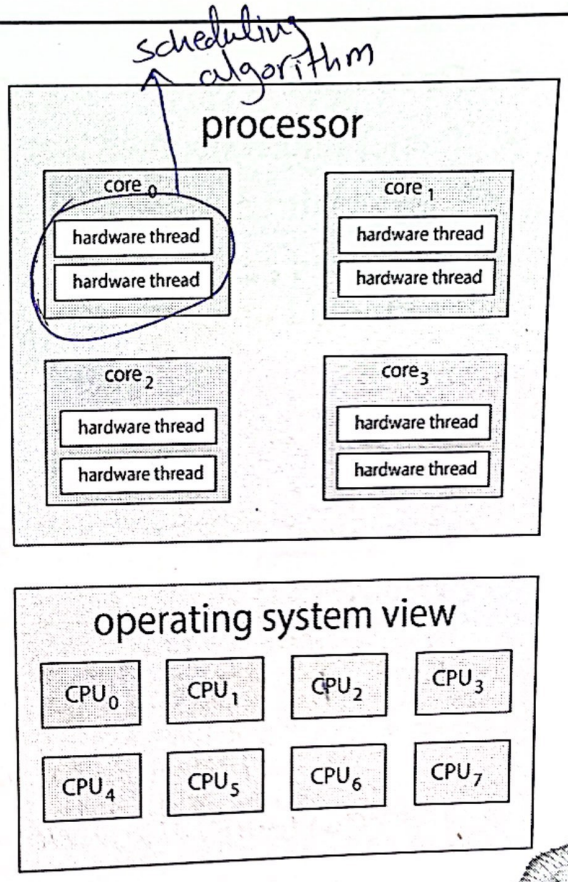
- On a quad-core system with 2 hardware threads per core, the operating system sees 8 logical processors.

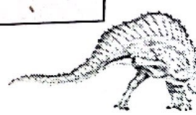
How many software threads can be scheduled on this Processor?

8 threads

↓

4 cores × 2 threads on each core.

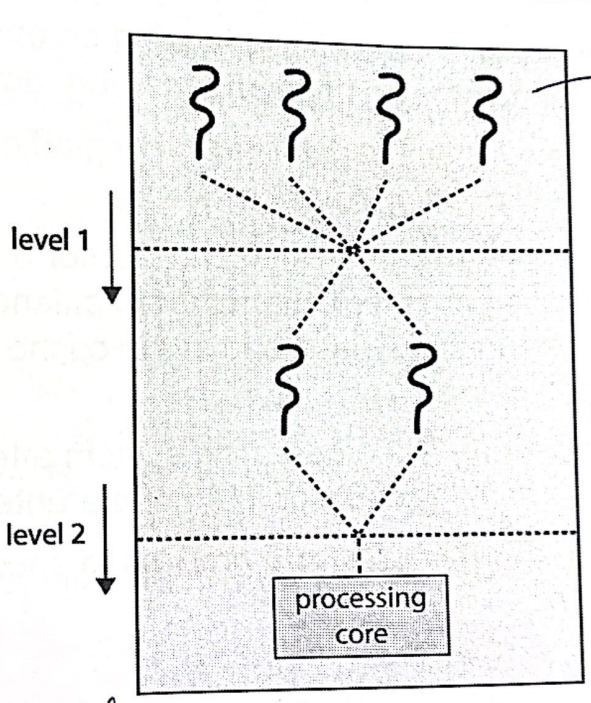




Multithreaded Multicore System

- Two levels of scheduling:

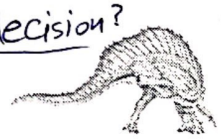
1. The operating system deciding which software thread to run on a logical CPU
2. How each core decides which hardware thread to run on the physical core.



Hyper threaded — scheduling levels

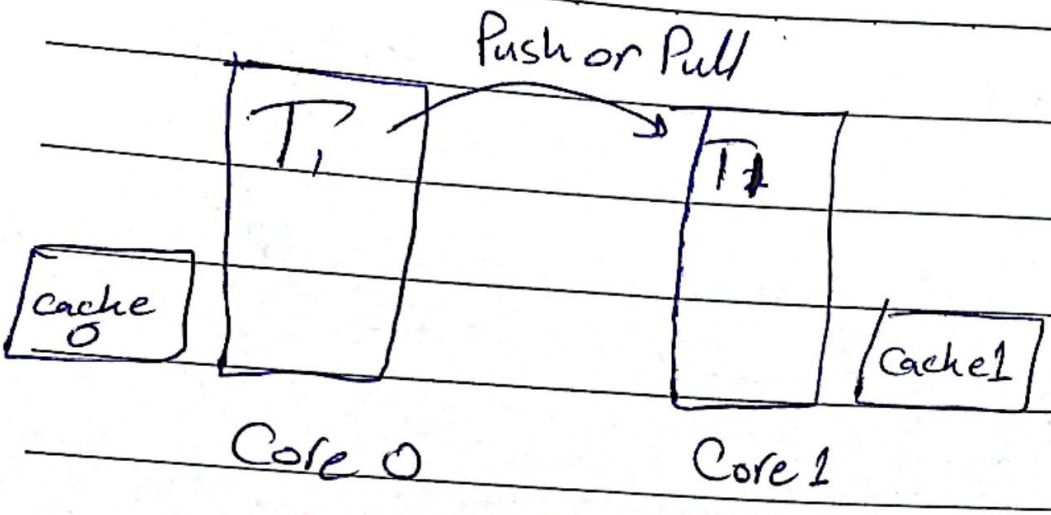
- many-to-many and HW threading, how many scheduling decision?

3 levels.



DATE

S M T W T F S



- Cache 0 is warm cache for T_1, and cache 1 is cold for T_1
- affinity of core 0 → which thread consider cache 0 as a warm cache. → T_1
- load balancing affects Processor affinity negatively

"end of chapter 5"

DATE

chapter 6 | slides (1-9) lecture 28

· Synchronization

Race Condition :- occurs when two threads or processes access a shared variable at the same time, and modify its value.

Atomic instruction :- cannot be interrupted while being executed.
(increment, decrement, are not atomic instructions)

P_0) Pid_t child = fork();
P_1) Pid_t child = fork();
P_2) Pid_t child = fork(); → next_available_id = 10
P_3) Pid_t child = fork();

Q) what is the minimum number of processes that will take the Pid = 10 ?
optimal ans :- 1 process. (Race condition may occur)

Q) what is the max. number of processes that will take the Pid = 10 ?
ans :- 4 processes. (worst case)

DATE S M T W T F S

~~Critical~~

- Critical section :- Part of the program which tries to access shared resources.
- The critical section cannot be executed by more than one process at the same time.

- entry section → each process requires permission to enter the critical section.

exit section → gives permission to the next process to enter the critical section.

Entry section (disable interrupts), so the process doesn't leave the critical section until it completes its task.

exit section (enable interrupts)

DATE

slides (9 - 22) lecture 29

critical section solutions:-

1- Interrupt-based solution → will cause some problems when the system is multicore, or the critical section code runs for long time.

2- Software solution → valid for two processes only.

3- Peterson's solution → valid for two processes only. uses "flag" variable to indicate if a process is ready to enter the critical section.

slide 17 → it will stuck in the while statment until the flag becomes true. (wait) / (busy wait)

. the expected output → x = 100

. In modern architecture there is reordering to memory accesses within the same thread (ex: inslide 17 → Print x before the while statment) →

DATE S M T W T F S

so the final expected output will be (x=0) if the execution started from Thread 1.

Example :- initially Flag[0] = F

P0 ① turn = 1;
Flag[0] = T;
while (Flag[1] && turn=1);
P0 will then enter CS

② P1 { turn = 0;
Flag[1] = T;
while (Flag[0] && turn==0); ③
P1 → is now in CS.

||

The result of reordering => invalid situation

Hardware supported solution (Memory Barrier)

write back → strongly ordered
write through → weakly ordered.

slides (32-39) lecture 30

- Mutex locks { 1 / True ; 0 / False } → boolean variable
- All critical sections that access the same shared resource, have the same locks.

example:- P_1 P_2 P_3

access (sh1) access (sh2) access (sh1)

P_1, P_3 have the same mutex locks because the access the same shared resource

ex:- (mutex_lock m_1;)

- When the lock equals 1 (ex:- $m_1 = 1$;), this means that this critical section is available (shared resource is available).

- acquire lock → $m_1 = 0$ / False
- release lock → $m_1 = 1$ / True.

```
acquire(m1) {
    while (!m1); → (when m1 = F, → busy wait).
    m1 = False; → The lock is now unavailable
    (critical section)
}
```

release (m,) {

m, = True; → The lock is available now.

}

• Busy wait :- technique in which a process repeatedly checks to see if a condition is true. (CPU is busy without any progress or Process).

So Mutex locks cause busy wait situation.

• Semaphore → used to avoid busy wait.

↳ use wait(), signal() instead of acquire, release to avoid busy wait.

• wait(S) { ← semaphore variable

while (S<=0); → the critical section is not available

wait queue ← // wait

until it becomes 1. S--; (if S=1 it will be 0 to protect the CS)

(critical section)

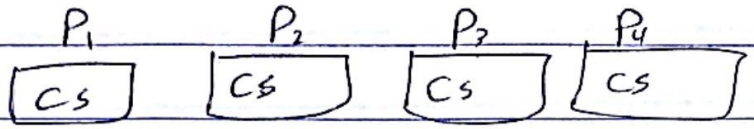
}

• Signal (S) {

S++ → S becomes 1 (from wait queue to ready queue)

}

• Counting semaphore : There are multiple instances of the shared resource (initial value for $S \neq 1$) → The initial value for S = # of available instances from the shared resource.



S = 4

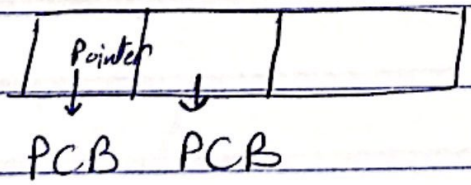
• How the busy wait problem was solved by using semaphore ?

- with each semaphore there is an associted waiting queue

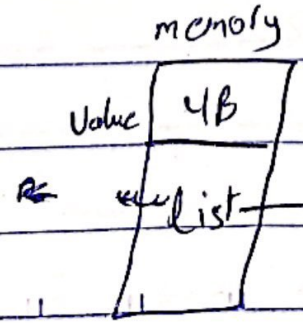
```
typedef struct {
    int value; → value of the semaphore
    struct process *list; → array of pointers, point the process.
} semaphore
```

(struct name)

• Wait queue contains the PCBs of waiting processes → Pointers to



Semaphore S;



DATE

S M T W T F S

Slides (34 - 44)

lecture 31

Semaphore
- binary semaphore
- Counting semaphore

integer variable

- Semaphore → when it is declared as variable not struct → it won't solve the busy wait problem

Slide 34.

```
wait(s) {
    while(s<=0);
    // busy wait
    s--;
}
signal(s) {
    s++;
}
```

may cause race condition.

→ So, (s-- & s++) should be atomic instructions (wait, signal) → atomic operations.

struct :- is a composite data type declaration that defines a physically grouped list of variables.

DATE S M T W T F S

slide 40

assume initially S=1, Value=1

~~Process started~~

$P_1 \rightarrow$ ① Value = 0 → ② CS

P_2 → ③ Value = -1 → ④ add the process to the wait list → ⑤ block.

P_3 → Value = -2 → Wait queue → block (delete from ready queue)

# of processes in the wait queue = |Value| = 2

P_1 → ① signal → ② Value = -1

P_2 → ③ wakeup → ④ ready queue

slides (45 - 58) lecture 32

. Condition Variables :- statements only (don't have values)
↳ data type :- Condition → ex:- Condition x;

. only two operations are allowed to execute on Condition Variables
↳ 1) x.wait() → suspended (wait) until another process execute x.signal()
(condition variable)

2) x.signal() → resumes only one process from processes that invoked x.wait()

ex:- P2 P1 → P2 before P1
(x.wait()) (x.signal()) ↳ Then P2 will wait until P1 executes x.signal(), then P2 will return to the ready queue.

. every (x.signal()) returns only one process from waiting queue to ready queue.

ex:- P1 : y.wait()
P2 :- y.wait()
P3 :- y.wait()
P4 :- y.signal()
P5 :- x.signal();

How many processes are in the waiting queue after executing y.signal()? 2 processes
How many processes are in the waiting queue? 2 processes.

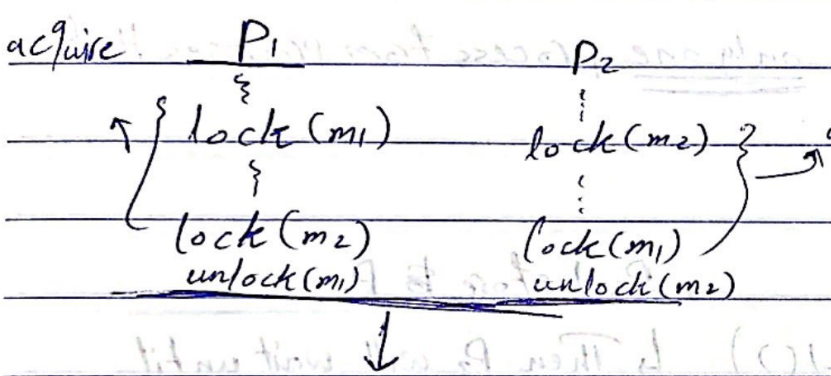
. each condition variable has its own wait queue.

/ /

- Deadlock :- situation in which the process have to wait for long time (unlimited), while trying to acquire a synchronization tool.

ex:- Critical section 1 → with lock (m_1)

" " 2 → with lock (m_2)



. P_1 will not continue to execute after it acquired lock (m_1) because lock(m_2) is already acquired by P_2, so P_1 will wait until P_2 release the lock. → Here P_1 will wait indefinitely because P_2 needs to acquire lock(m_1) to release lock(m_2) which is already acquired by P_1 → so, P_1 and P_2 will wait indefinitely.

Deadlock. → liveness failure.

- liveness failure another example → infinite loop.

slides (1-6)

Chapter 7 | synchronization examples

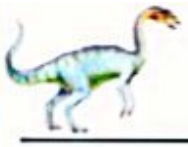
Synchronization problems → 1) bounded-buffer
2) readers-writers.

• Bounded-buffer → if there is at least one writer process at the buffer that will modify the original stored value, other writer and reader processes are not allowed to use the buffer. → using synchron. tools. → we need to create semaphore mutex (value = 1), and count semaphore called semaphore full (value = 0) which represents the number of full buffers. Also, semaphore empty which represents number of empty buffers (value = # of buffers).

slide (5) ↘

• Assumption → the producer will make the buffer full, and the consumer will empty it.

• when mutex = 1 → the process can enter the critical section.
↳ after executing (~~mutex~~ wait(mutex), mutex becomes zero.
↳ then signal(mutex) returns the value of the mutex to become = 1
↳ signal(full) → increment.



Bounded Buffer Problem (Cont.)

- The structure of the producer process

```
while (true) {
        ...
        /* produce an item in next_produced */
        ...
    wait(empty);
    wait(mutex);
        ...
        /* add next produced to the buffer */
        ...
    signal(mutex);
    signal(full);
}
```

DECREMENT THE VALUE OF EMPTY

WAIT ONLY WHEN EMPTY = 0 (WHEN ALL BUFFERS ARE FULL)

THE PRODUCER WILL ENTER THE CRITICAL SECTION (WRITING ON THE BUFFER).

MUTEX = 1 AGAIN

INCREMENT (FULL ++)

Bounded Buffer Problem (Cont.)

- The structure of the consumer process

```
while (true) {
    wait(full);     DECREMENT (FULL = N-1)
    wait(mutex);    MUTEX BECOMES ZERO
       ...
    /* remove an item from buffer to next_consumed */
       ...
    signal(mutex);  MUTEX = 1
    signal(empty);  INCREMENT EMPTY ++
       ...
    /* consume the item in next consumed */
       ...
}
```

lecture 33

slides (7-15)

• Readers-Writers Problem → only one single writer can access the shared data at the same time. (without any readers or writers)

• first reader writer variation → if we have on reader in the shared area, all incoming readers are allowed to use the shared area. → which cause starvation (writers) ↑ at least

• second reader writer variation → once a writer is ready, that writer performs its write as soon as possible (no new readers may start reading). even if there are readers in the shared area. → which cause starvation (readers) ↑

slide 8)

read_count → represents # of readers in the shared area → if read_count != 1 new coming readers can use the shared area without acquiring the lock. only the first reader acquire the lock to prevent the writers only from using the shared area.

• The last reader leaving the shared area release the lock ↳ when (read_count = 0)

• read_count → shared variable (may cause race condition)

slides (16-18) lecture 34

- difference between named and unnamed semaphores → declaration

remember :- Condition variables are user-mode objects that cannot be shared between processes → used to make sure that the shared resource is accessed by one process at a time → so mutex or binary semaphore is used with the condition variables.

slide 18) a, b → shared resources

~~pthread_mutex_unlock()~~

pthread_cond_wait(&cond_var, &mutex);

↳ to protect a, b

(release the lock)

Chapter 8 | slides (1-6)

remember → Deadlocks :- situation in which the process have to wait for long time, while trying to acquire a synchronization tool.

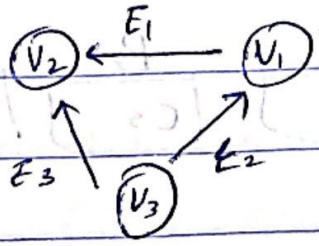
. CPU utilization in deadlock situation = 0%.

. Deadlock characterization → formal definition.

slides (7 - 12) lecture 35

• Resource - Allocation Graph. → group of nodes (vertices) and directed eagdes.

ex:-



• request edge (from processes or threads to a resource)
• assignment edge (from a dot in the resource to a thread or process)
• if there is a cycle in the graph (The starting point is the same as the end point), There may be a deadlock situation or not, but if there isn't any cycle in the graph then there is no any deadlock situation.

slide 9

$T_2 \rightarrow R_3 \rightarrow T_3 \rightarrow R_2 \rightarrow T_2$

↓

cycle, is there any dead lock?

• T_3 acquires R_3, and needs R_2
• T_2 acquires R_2, and needs R_3

↓

Deadlock situation.

slide 9

$T_1 \rightarrow R_1 \rightarrow T_2 \rightarrow R_3 \rightarrow T_3 \rightarrow R_2 \rightarrow T_1$

- T_1 acquires R_2, and needs R_1
- T_2 acquires R_1, and needs R_3
- T_2 acquires R_2, and needs R_3
- T_3 acquires R_3, and needs R_2

↓

T_1, T_2, T_3 Deadlock

- If there is only one instance from resources that are involved within the cycle → 100% there is a deadlock situation for all threads.

---

slide 10

$T_1 \rightarrow R_1 \rightarrow T_3 \rightarrow R_2 \rightarrow T_1$

- T_1 acquires R_2, and needs R_1
- T_3 acquires R_1, and needs R_2
- if T_4 release R_2 → no deadlock (break the cycle)

- Deadlock prevention → By preventing the occurrence of the 4 deadblock conditions at the same time.

- Deadlock avoidance :- when the Kernel has Previous Knowledge about future requests of resources.

Chapter 9 | slides (4 - 6)

- CPU can only access Main memory and registers
- read, write operations take 1 CPU cycle or less. (registers)
- read, write from/on the main memory is slower than accessing registers. (more cycles are needed), so cache is used and sits between main memory and CPU registers.

- base address → The beginning of the address space
- base + limit → where the limit is the size of the address space, so (limit + base) the end of the address space.

- The process itself can't change the base and the limit because they are Privileged.

base ≤ logical address space of a process < base + limit

Good Luck!