

CS330

Operating Systems

PROCESS (I)

Prof. Dr. Basit Qureshi
PhD FHEA SMIEEE MACM
www.drbasit.org

TOPICS

- The concept of a Process
- Operations in a process
- Inter-process communication
 - Shared memory
 - Message passing

Credit:

These notes are extracted from the following resources:

- Silberschatz, Galvin and Gagne, *Operating System Concepts*, 10 ed, Wiley, 2018
- Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau, *Operating Systems: Three Easy Pieces*, CreateSpace, 2018

CS330

Operating Systems

PROCESS CONCEPT

Prof. Dr. Basit Qureshi
PhD FHEA SMIEEE MACM
www.drbasit.org

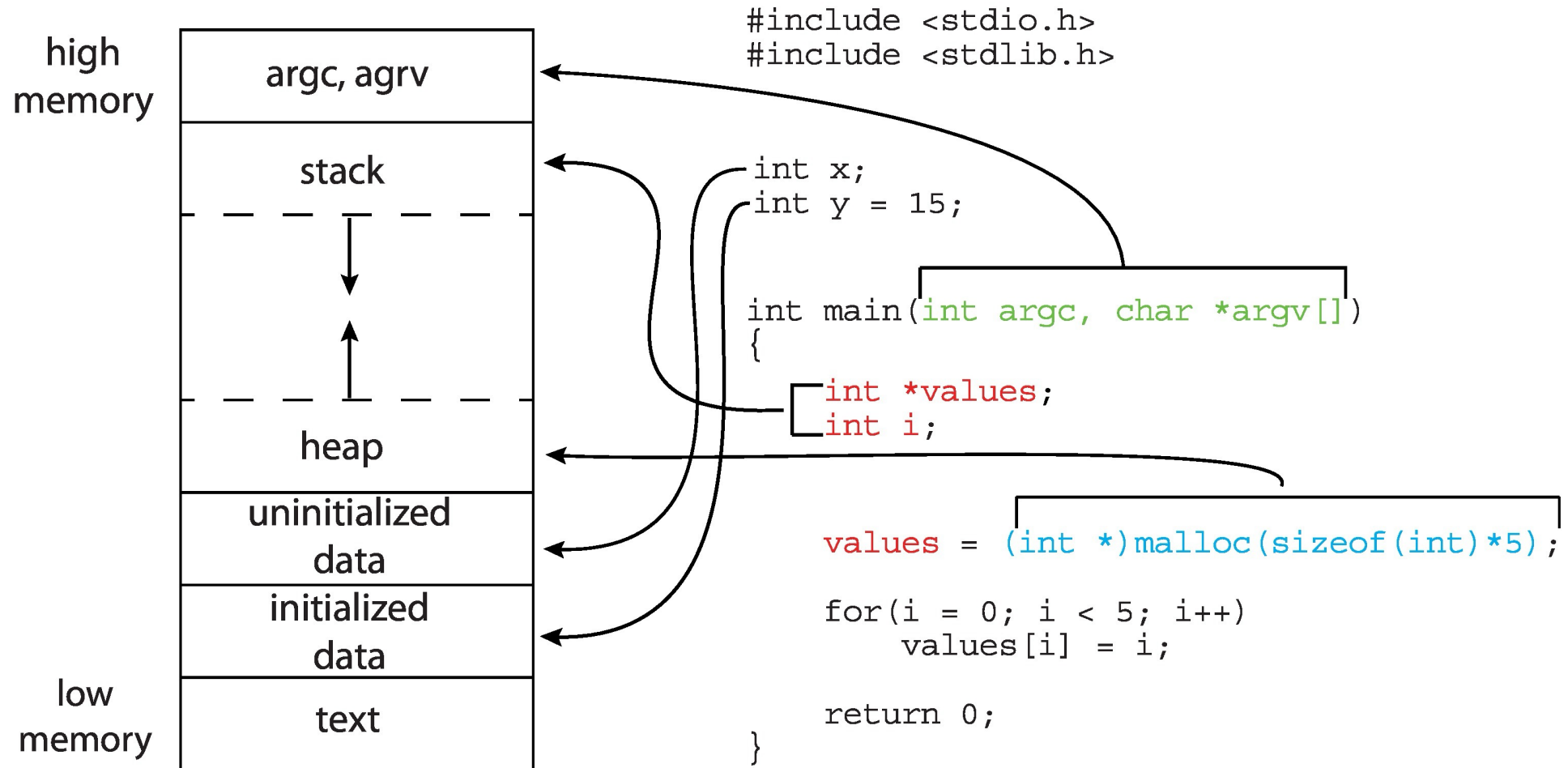
PROCESS CONCEPT

- An operating system executes a variety of programs that run as a process.
- **Process** – a program in **execution**; *process execution must progress in sequential fashion*. No parallel execution of instructions of a single process
- Multiple parts
 - The program code, also called **text section**
 - Current activity including **program counter**, processor **registers**
 - **Stack** containing temporary data
 - Function parameters, return addresses, local variables
 - **Data section** containing global variables
 - **Heap** containing memory *dynamically* allocated during run time

PROCESS CONCEPT (CONT.)

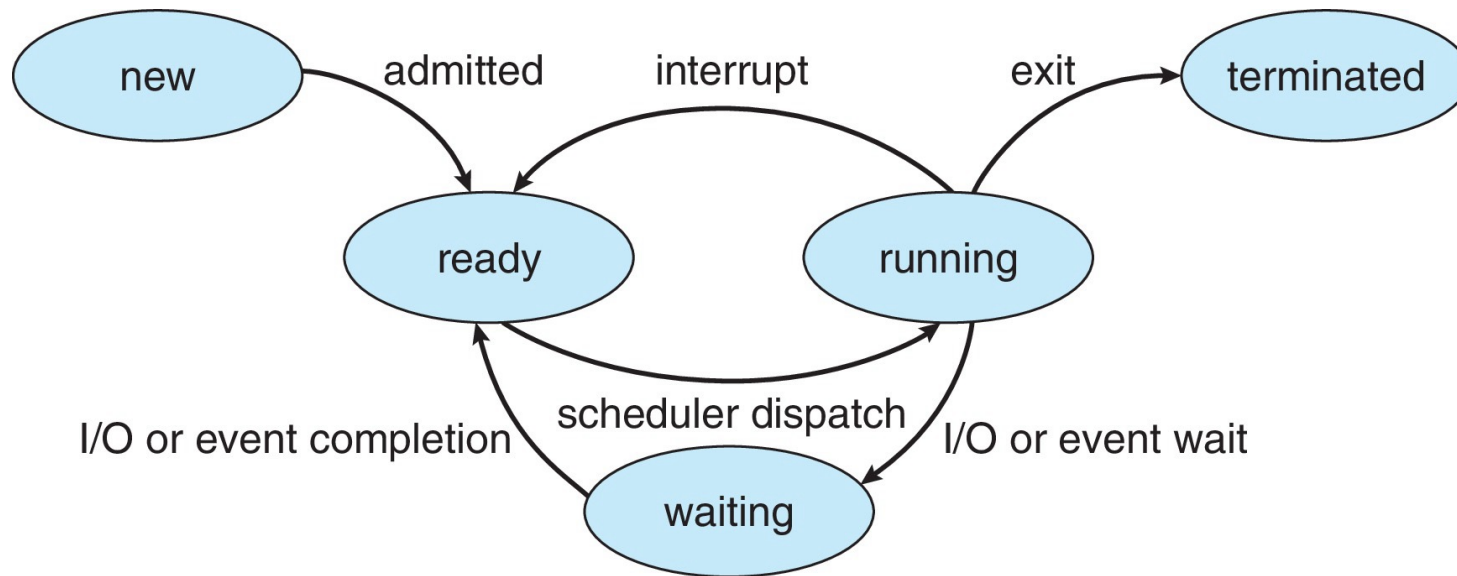
- Program is **passive** entity stored on disk (**executable file**); process is **active**. **Program** becomes **process** when an executable file is loaded into memory
- One program can be several processes
 - Consider multiple users executing the same program

MEMORY LAYOUT OF A C PROGRAM (PROCESS)



PROCESS STATE

- As a process executes, it changes **state**
 - **New**: The process is being created
 - **Running**: Instructions are being executed
 - **Waiting**: The process is waiting for some event to occur
 - **Ready**: The process is waiting to be assigned to a processor
 - **Terminated**: The process has finished execution



PROCESS CONTROL BLOCK (**PCB**)

- Information associated with each process(also called **task control block**)
- **Process state** – running, waiting, etc.
- **Program counter** – location of instruction to next execute
- **CPU registers** – contents of all process-centric registers
- **CPU scheduling information**- priorities, scheduling queue pointers
- **Memory-management information** – memory allocated to the process
- **Accounting information** – CPU used, clock time elapsed since start, time limits
- **I/O status information** – I/O devices allocated to process, list of open files

process state
process number
program counter
registers
memory limits
list of open files
...

THREADS

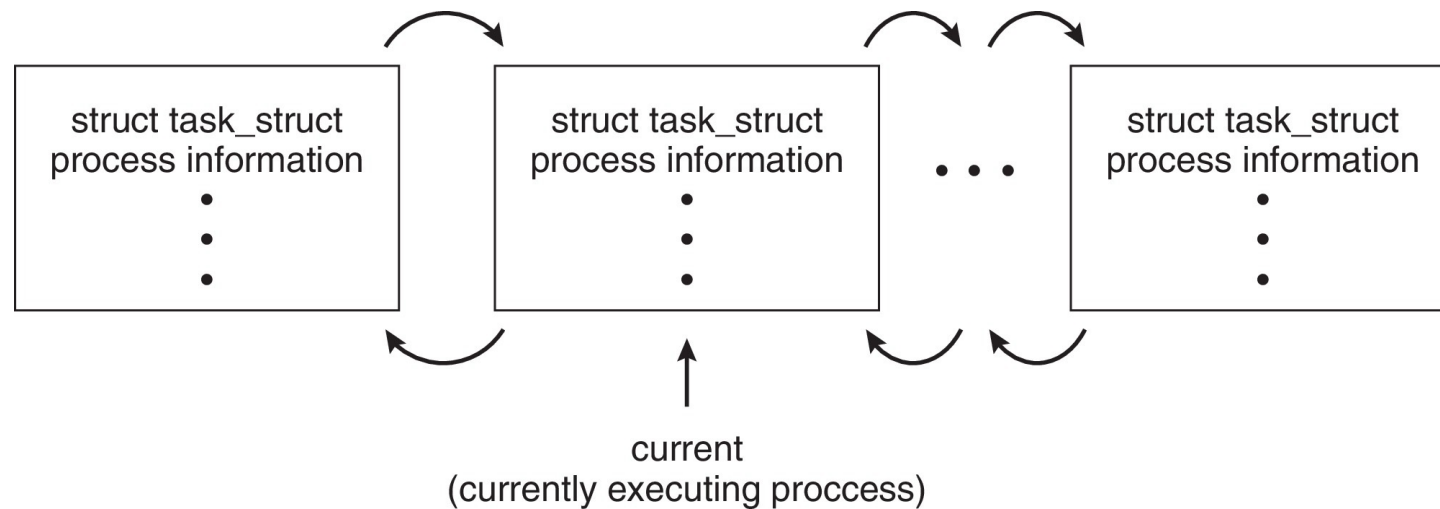
Explored later

- So far, process has a single thread of execution
- Consider having multiple program counters per process
 - Multiple locations can execute at once
 - Multiple threads of control -> **threads**
- Must then have storage for thread details, multiple program counters in PCB

PROCESS REPRESENTATION IN LINUX

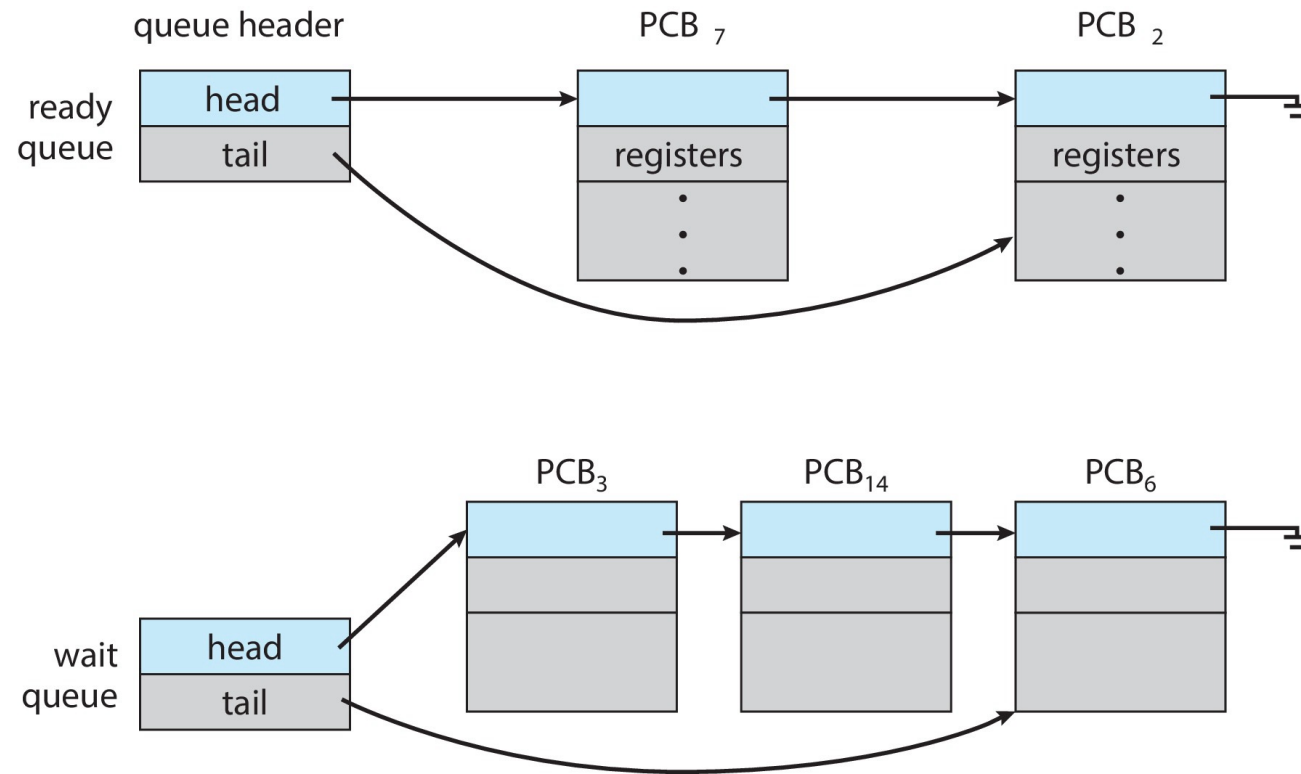
Represented by the C structure `task_struct`

```
pid t_pid;           /* process identifier */
long state;          /* state of the process */
unsigned int time_slice /* scheduling information */
struct task_struct *parent; /* this process's parent */
struct list_head children; /* this process's children */
struct files_struct *files; /* list of open files */
struct mm_struct *mm;    /* address space of this process */
```

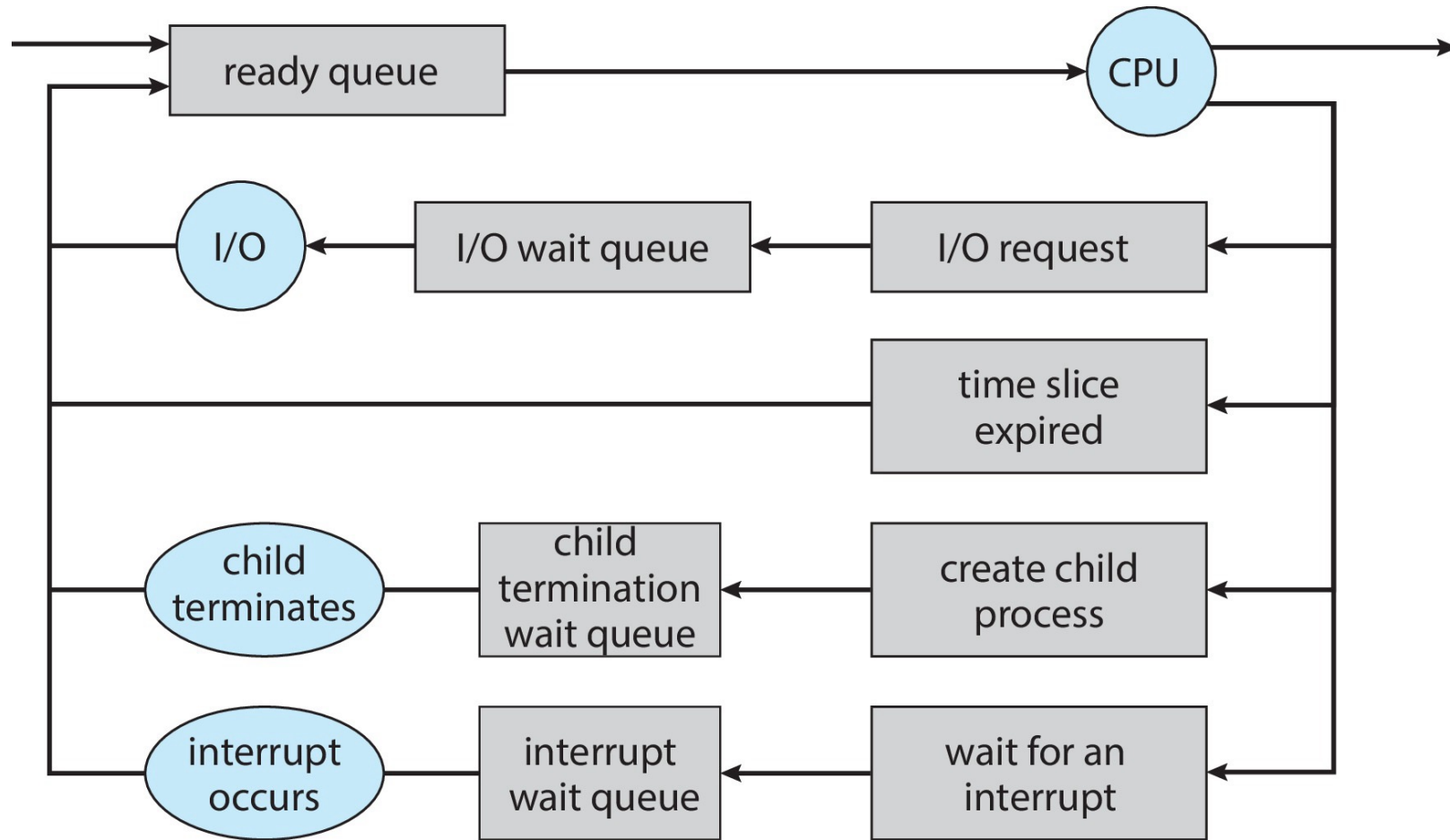


PROCESS SCHEDULING

- **Process scheduler** selects available processes for next execution in CPU
- Goal -- **Maximize CPU use**, quickly switch processes onto CPU core
- Maintains **scheduling queues** of processes
 - **Ready queue** – set of all processes in main memory, ready and waiting to execute
 - **Wait queues** – set of processes waiting for an event (i.e., I/O)



REPRESENTATION OF PROCESS SCHEDULING

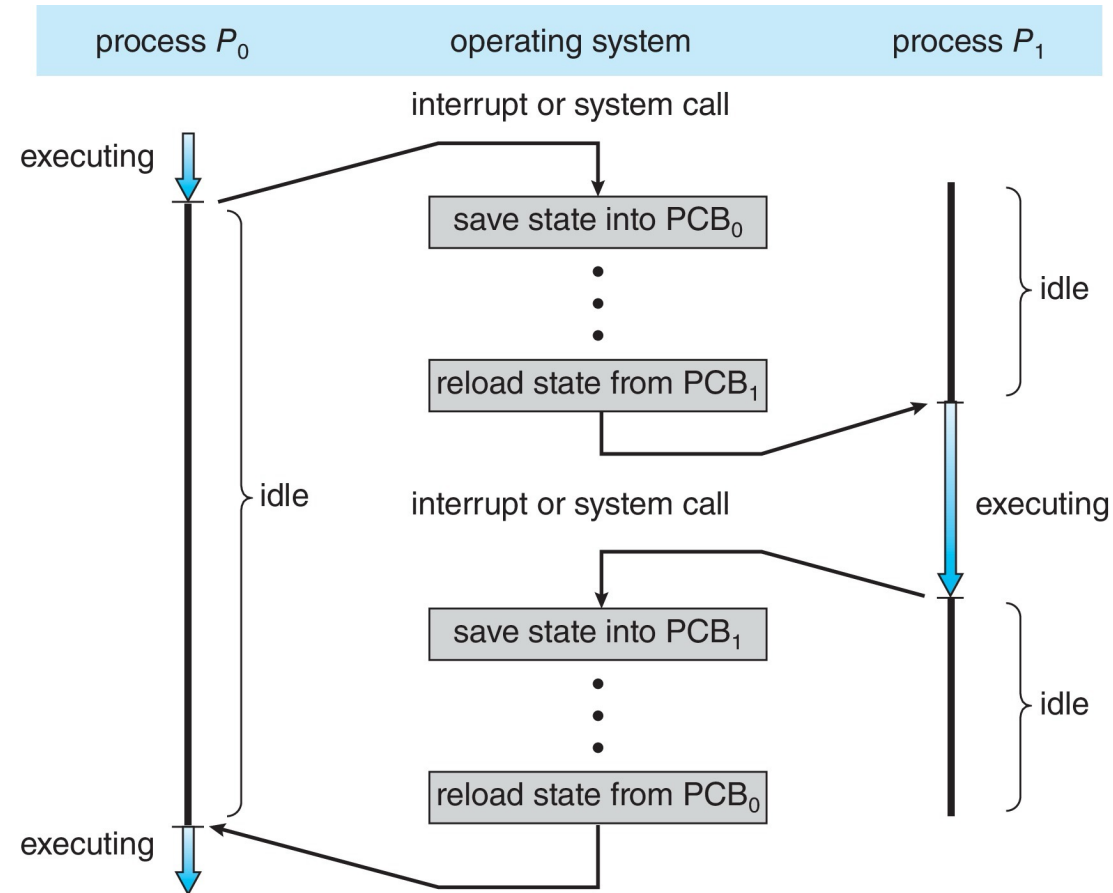


CONTEXT SWITCH

- When CPU switches to another process, the system must **save the state** of the old process and load the **saved state** for the new process via a **context switch**
- **Context** of a process represented in the PCB
- Context-switch time is ***pure overhead***; the system does no useful work while switching
 - The more complex the OS and the PCB → the longer the context switch
- Time dependent on hardware support
 - Some hardware provides multiple sets of registers per CPU → multiple contexts loaded at once

CPU SWITCH FROM PROCESS TO PROCESS

A **context switch** occurs when the CPU switches from one process to another.



CS330

Operating Systems

OPERATIONS IN A PROCESS

Prof. Dr. Basit Qureshi
PhD FHEA SMIEEE MACM
www.drbasit.org

OPERATIONS ON PROCESSES

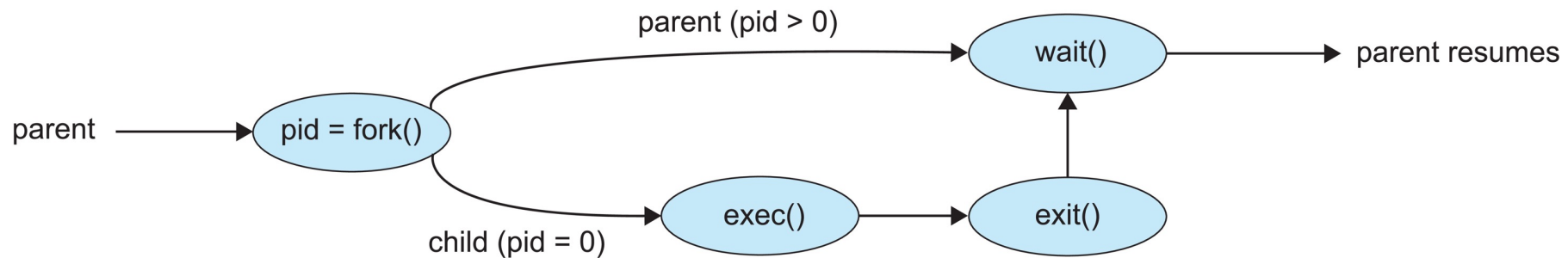
- System must provide mechanisms for:
 - Process **creation**
 - Process **termination**

PROCESS CREATION

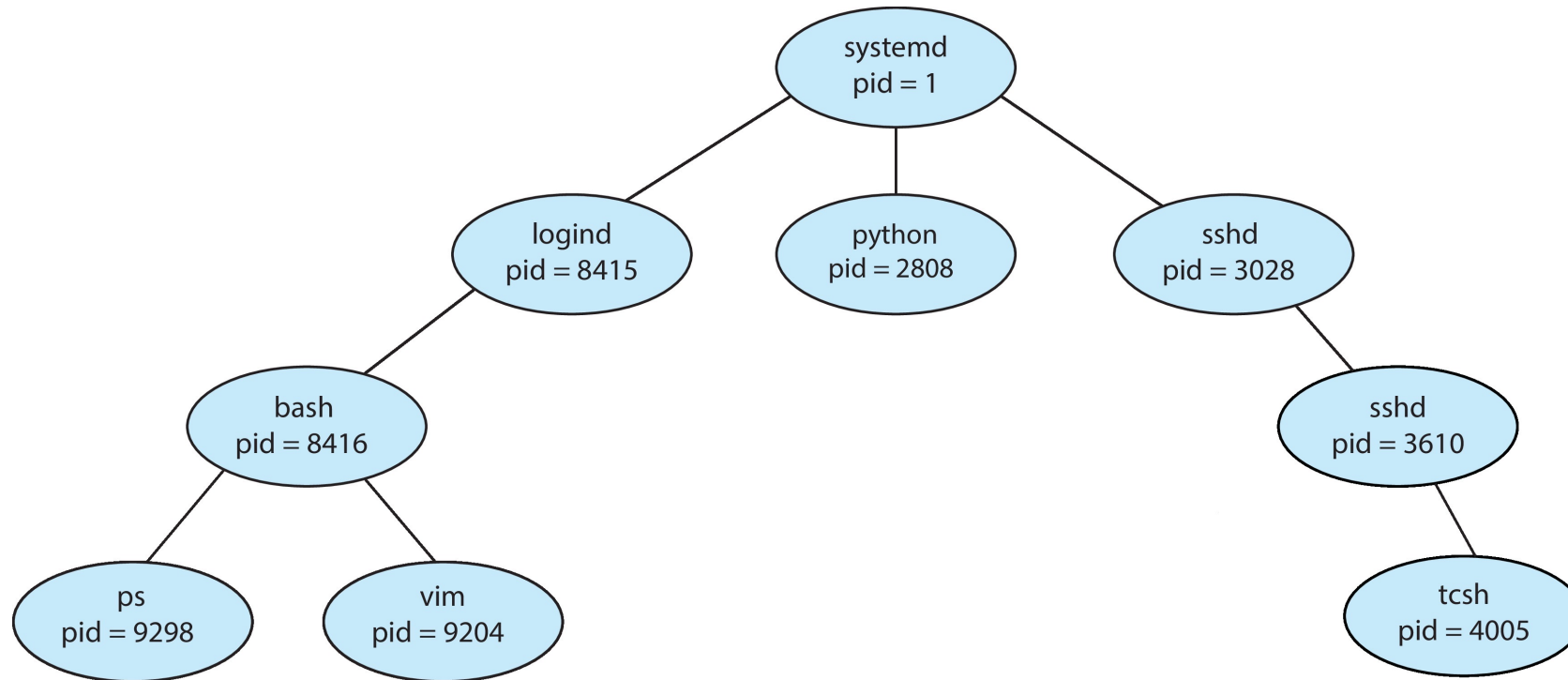
- **Parent** process create **children** processes, which, in turn create other processes, forming a **tree** of processes
- Generally, process identified and managed via a **process identifier (pid)**
- Resource sharing options
 1. Parent and children share all resources
 2. Children share subset of parent's resources
 3. Parent and child share no resources
- Execution options
 1. Parent and children execute concurrently
 2. Parent waits until children terminate

PROCESS CREATION (CONT.)

- Address space
 - Child duplicate of parent
 - Child has a program loaded into it
- UNIX examples
 - **fork()** system call creates new process
 - **exec()** system call used after a **fork()** to replace the process' memory space with a new program
 - Parent process calls **wait()** waiting for the child to terminate



A TREE OF PROCESSES IN LINUX



C PROGRAM FORKING SEPARATE PROCESS

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
    pid_t pid;

    /* fork a child process */
    pid = fork();

    if (pid < 0) { /* error occurred */
        fprintf(stderr, "Fork Failed");
        return 1;
    }
    else if (pid == 0) { /* child process */
        execlp("/bin/ls", "ls", NULL);
    }
    else { /* parent process */
        /* parent will wait for the child to complete */
        wait(NULL);
        printf("Child Complete");
    }

    return 0;
}
```

CREATING A SEPARATE PROCESS VIA WINDOWS API

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

int main(VOID)
{
    STARTUPINFO si;
    PROCESS_INFORMATION pi;

    /* allocate memory */
    ZeroMemory(&si, sizeof(si));
    si.cb = sizeof(si);
    ZeroMemory(&pi, sizeof(pi));

    /* create child process */
    if (!CreateProcess(NULL, /* use command line */
        "C:\\WINDOWS\\system32\\mspaint.exe", /* command */
        NULL, /* don't inherit process handle */
        NULL, /* don't inherit thread handle */
        FALSE, /* disable handle inheritance */
        0, /* no creation flags */
        NULL, /* use parent's environment block */
        NULL, /* use parent's existing directory */
        &si,
        &pi))
    {
        fprintf(stderr, "Create Process Failed");
        return -1;
    }
    /* parent will wait for the child to complete */
    WaitForSingleObject(pi.hProcess, INFINITE);
    printf("Child Complete");

    /* close handles */
    CloseHandle(pi.hProcess);
    CloseHandle(pi.hThread);
}
```

PROCESS TERMINATION

- Process executes last statement and then asks the operating system to delete it using the `exit()` system call.
 - Returns status data from child to parent (via `wait()`)
 - Process' resources are deallocated by operating system
- Parent may terminate the execution of children processes using the `abort()` system call. Some reasons for doing so:
 - Child has exceeded allocated resources
 - Task assigned to child is no longer required
 - The parent is exiting, and the operating systems does not allow a child to continue if its parent terminates

PROCESS TERMINATION

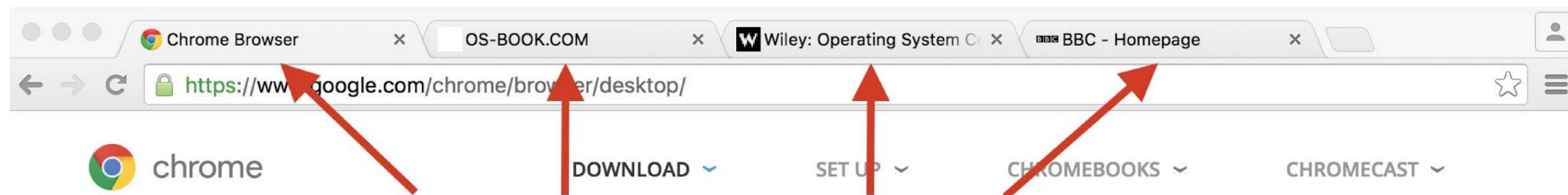
- Some operating systems **do not allow** child to **exist** if its parent has **terminated**. *If a process terminates, then all its children must also be terminated.*
 - **Cascading termination.** All children, grandchildren, etc., are terminated.
 - The termination is initiated by the operating system.
- The parent process may wait for termination of a child process by using the **wait()** system call. The call returns status information and the pid of the terminated process

```
pid = wait(&status) ;
```

- If no parent waiting (did not invoke **wait()**) process is a **zombie**
- If parent terminated without invoking **wait()**, process is an **orphan**

MULTIPROCESS ARCHITECTURE – CHROME BROWSER

- Many web browsers ran as single process (some still do)
 - If one web site causes trouble, entire browser can hang or crash
- Google Chrome Browser is multiprocess with 3 different types of processes:
 - **Browser** process manages user interface, disk and network I/O
 - **Renderer** process renders web pages, deals with HTML, Javascript. A new renderer created for each website opened
 - Runs in **sandbox** restricting disk and network I/O, minimizing effect of security exploits
 - **Plug-in** process for each type of plug-in



Each tab represents a separate process.

CS330

Operating Systems

INTER-PROCESS COMMUNICATION

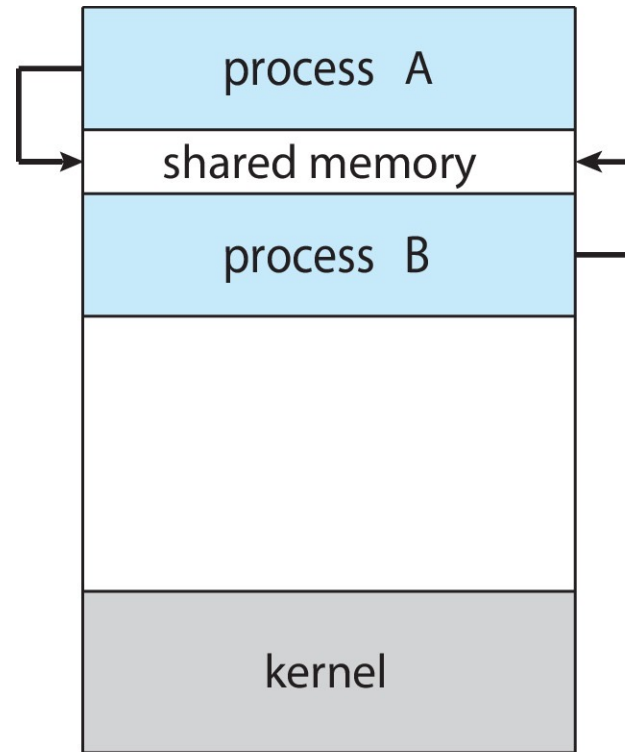
Prof. Dr. Basit Qureshi
PhD FHEA SMIEEE MACM
www.drbasit.org

INTERPROCESS COMMUNICATION

- Processes within a system may be ***independent*** or ***cooperating***
- Cooperating process can affect or be affected by other processes, *including sharing data*
- Reasons for cooperating processes:
 - Information sharing
 - Computation speedup
 - Modularity
 - Convenience
- Cooperating processes need **interprocess communication (IPC)**
- Two models of IPC
 - **Shared memory**
 - **Message passing**

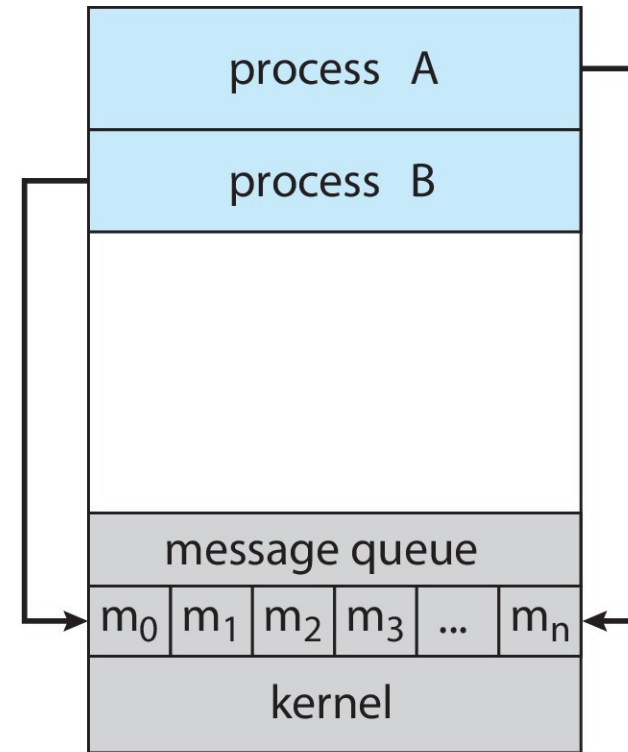
COMMUNICATIONS MODELS

(a) Shared memory.



(a)

(b) Message passing.



(b)

CS330

Operating Systems

IPC: SHARED MEMORY

Prof. Dr. Basit Qureshi
PhD FHEA SMIEEE MACM
www.drbasit.org

PRODUCER-CONSUMER PROBLEM

- Paradigm for cooperating processes:
 - ***producer*** process produces information that is consumed by a ***consumer*** process
- Two variations:
 - **unbounded-buffer** places no practical limit on the size of the buffer:
 - Producer never waits
 - Consumer waits if there is no buffer to consume
 - **bounded-buffer** assumes that there is a fixed buffer size
 - Producer must wait if all buffers are full
 - Consumer waits if there is no buffer to consume

IPC – SHARED MEMORY

- An area of memory shared among the processes that wish to communicate
- The communication is under the control of the users processes not the operating system.
- Major issues is to provide mechanism that will allow the user processes to synchronize their actions when they access shared memory.
- Synchronization is discussed in great details in Chapters 6 & 7.

BOUNDED-BUFFER – SHARED-MEMORY SOLUTION

- Shared data

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

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

- Solution is correct, but can only use **BUFFER_SIZE-1** elements

PRODUCER PROCESS – SHARED MEMORY

```
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;
}
```

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 */
}
```

WHAT ABOUT FILLING ALL THE BUFFERS?

- Suppose that we wanted to provide a solution to the consumer-producer problem that fills **all** the buffers.
- We can do so by having an integer `counter` that keeps track of the number of full buffers.
- Initially, `counter` is set to 0.
- The integer `counter` is incremented by the producer after it produces a new buffer.
- The integer `counter` is and is decremented by the consumer after it consumes a buffer.

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++;  
}
```

CONSUMER

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

RACE CONDITION

- **counter++** could be implemented as

```
register1 = counter
register1 = register1 + 1
counter = register1
```

- **counter--** could be implemented as

```
register2 = counter
register2 = register2 - 1
counter = register2
```

- Consider this execution interleaving with “count = 5” initially:

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

RACE CONDITION (CONT.)

- Question – why was there no race condition in the first solution (where at most $N - 1$) buffers can be filled?
- More in Chapter 6.

CS330

Operating Systems

IPC: MESSAGE PASSING

Prof. Dr. Basit Qureshi
PhD FHEA SMIEEE MACM
www.drbasit.org

IPC – MESSAGE PASSING

- Processes communicate with each other without resorting to shared variables
- IPC facility provides two operations:
 - **send**(*message*)
 - **receive**(*message*)
- The *message* size is either fixed or variable

MESSAGE PASSING (CONT.)

- If processes P and Q wish to communicate, they need to:
 - Establish a ***communication link*** between them
 - Exchange messages via send/receive
- Implementation issues:
 - How are links established?
 - Can a link be associated with more than two processes?
 - How many links can there be between every pair of communicating processes?
 - What is the capacity of a link?
 - Is the size of a message that the link can accommodate fixed or variable?
 - Is a link unidirectional or bi-directional?

IMPLEMENTATION OF COMMUNICATION LINK

- Physical:
 - Shared memory
 - Hardware bus
 - Network
- Logical:
 - Direct or indirect
 - Synchronous or asynchronous
 - Automatic or explicit buffering

DIRECT COMMUNICATION

- Processes must name each other explicitly:
 - **send** (P , *message*) – send a message to process P
 - **receive**(Q , *message*) – receive a message from process Q
- Properties of communication link
 - Links are established automatically
 - A link is associated with exactly one pair of communicating processes
 - Between each pair there exists exactly one link
 - The link may be unidirectional, but is usually bi-directional

INDIRECT COMMUNICATION

- Messages are directed and received from mailboxes (also referred to as ports)
 - Each mailbox has a unique id
 - Processes can communicate only if they share a mailbox
- Properties of communication link
 - Link established only if processes share a common mailbox
 - A link may be associated with many processes
 - Each pair of processes may share several communication links
 - Link may be unidirectional or bi-directional

INDIRECT COMMUNICATION (CONT.)

- Operations
 - Create a new mailbox (port)
 - Send and receive messages through mailbox
 - Delete a mailbox
- Primitives are defined as:
 - **send**(*A, message*) – send a message to mailbox A
 - **receive**(*A, message*) – receive a message from mailbox A

INDIRECT COMMUNICATION (CONT.)

- Mailbox sharing
 - P_1 , P_2 , and P_3 share mailbox A
 - P_1 , sends; P_2 and P_3 receive
 - Who gets the message?
- Solutions
 - Allow a link to be associated with at most two processes
 - Allow only one process at a time to execute a receive operation
 - Allow the system to select arbitrarily the receiver. Sender is notified who the receiver was.

PRODUCER-CONSUMER: MESSAGE PASSING

- Producer

```
message next_produced;  
while (true) {  
    /* produce an item in next_produced */  
  
    send(next_produced) ;  
}
```

- Consumer

```
message next_consumed;  
while (true) {  
    receive(next_consumed)  
  
    /* consume the item in next_consumed */  
}
```