Lecture 02
Process Management

Dr. Ehtesham Zahoor

Some review

- How distributed systems are related to the operating systems?
- What is Cloud computing and the virtualization?
- How blockchain is related to our course?

Operating Systems Research

- This year, the course project would involve writing term paper and presentation
- Some words about the last year project

Process

- A computer program in execution on a machine is a process

- More formally:
  - A **Sequential stream of Execution** in its own **address space**

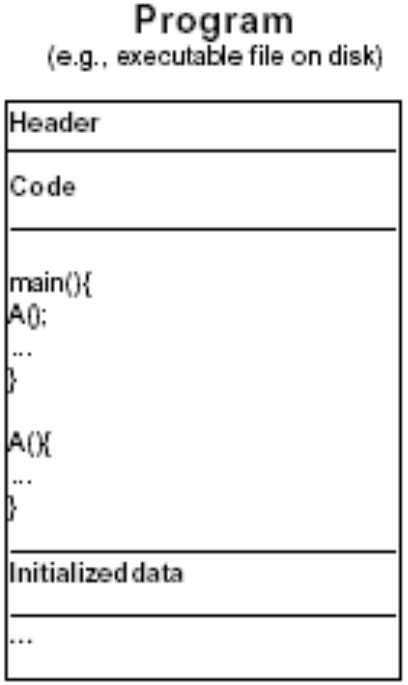
Why use processes?

- Why use processes?
  - Express concurrency
  - Systems have many concurrent jobs going on
  - E.g. Multiple users running multiple shells, I/O, ...

- General principle of divide and conquer
  - Decompose a large problem into smaller ones
  - easier to think of well contained smaller problems

Process =? Program

- Program: series of commands (e.g. C statements, assembly commands, shell commands)



Program != process

- Program: static code + static data
- Process: dynamic instantiation of code + data + more

- Program process: no 1:1 mapping
  - Process has more than code and data
  - one program runs many processes
  - many processes of same program

Process

- A computer program in execution on a machine is a process

- More formally:
  - A **Sequential stream of Execution** in its own **address space**

Process Address Space

- A list of memory locations from some min to some max that a process can read and write.

- Contains
  - the executable program
  - program's data
  - Stack
  - Associated with a process is a set of registers e.g. PC,SP and other information to run the program.

CPU State

- CPU registers contain the current state

  Instruction Register (IR):

  Program Counter (PC):

  Stack Pointer (SP):

  General purpose registers:

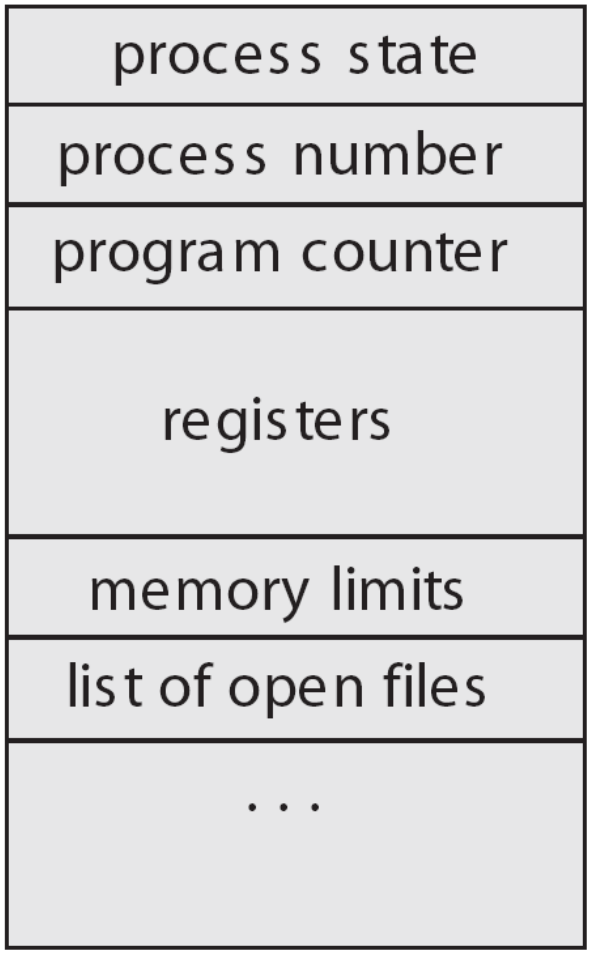
  …

Environment

- Contains the relationships with other entities
- A process does not exist in a vacuum
- It typically has connections with other entities, such as
  - A terminal where the user is sitting.
  - Open files
  - Communication channels to other processes, possibly on other machines.

Process Control Block

- The OS keeps all the data it needs about a process in the process control block (PCB)



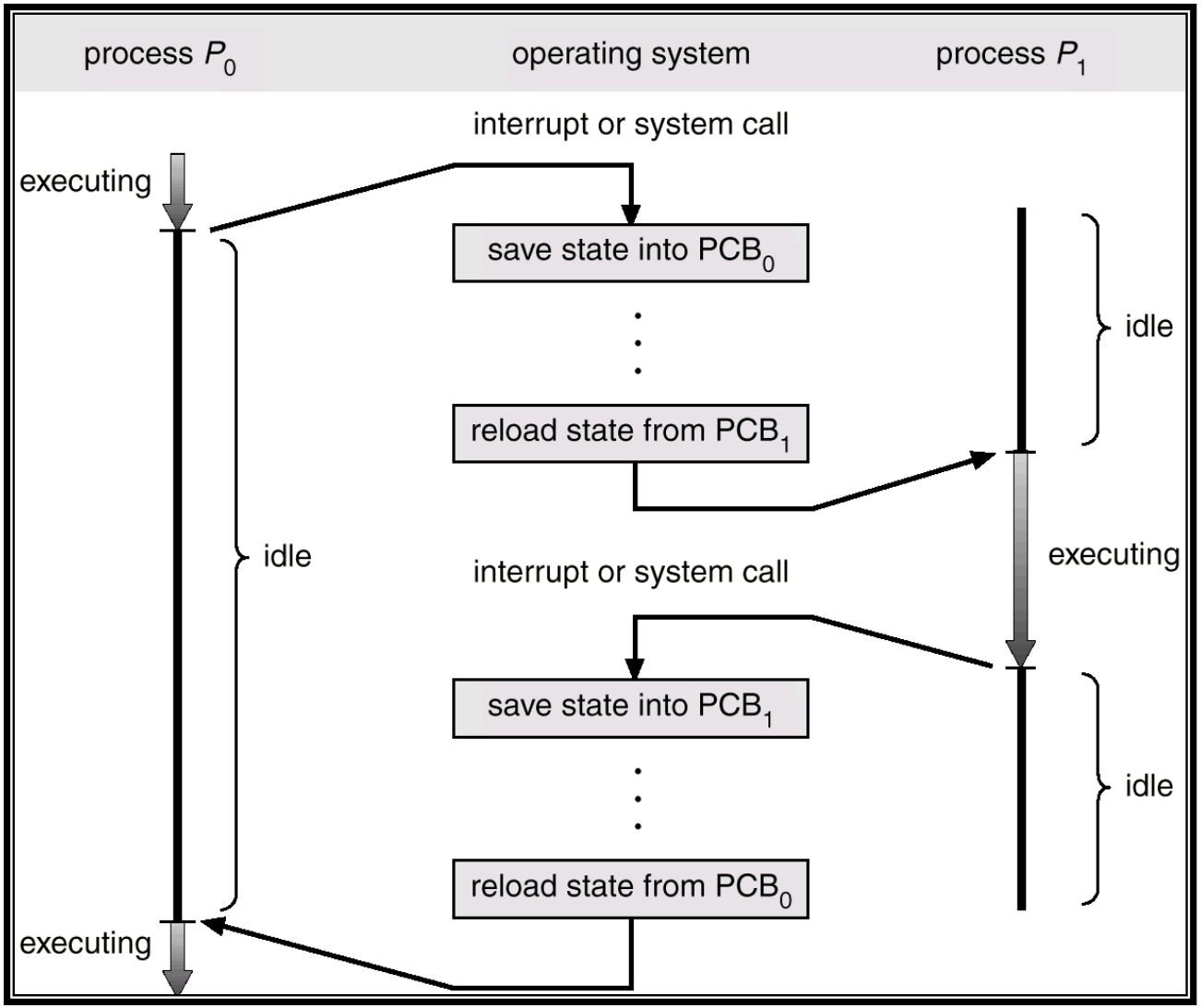
PCB

- PCB is "the manifestation of a process in an operating system”.
- **Data Structure** defined in the operating system kernel containing the information needed to manage a particular process.
- It must be kept in an area of memory protected from normal user access.

PCB

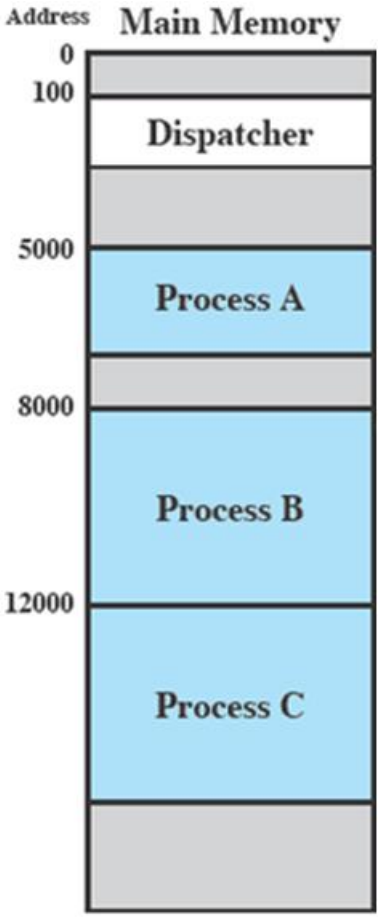
- In general a PCB includes:
  - The identifier of the process (a process identifier, or PID)
  - Register values for the process including, notably, the program counter and stack pointer values for the process.
  - The address space for the process
  - Priority (in which higher priority process gets first preference. e.g., nice value on Unix operating systems)
  - Process accounting information, such as when the process was last run, how much CPU time it has accumulated, etc.
  - I/O Information (i.e. I/O devices allocated to this process, list of opened files, etc.)

CPU Switch From Process to Process



CPU Switch From Process to Process

- Switching a process requires
  - Saving the state of old process
  - Loading the saved state of the new process
- This is called **Context Switch**
- Part of OS responsible for switching the processor among the processes is called **Dispatcher**

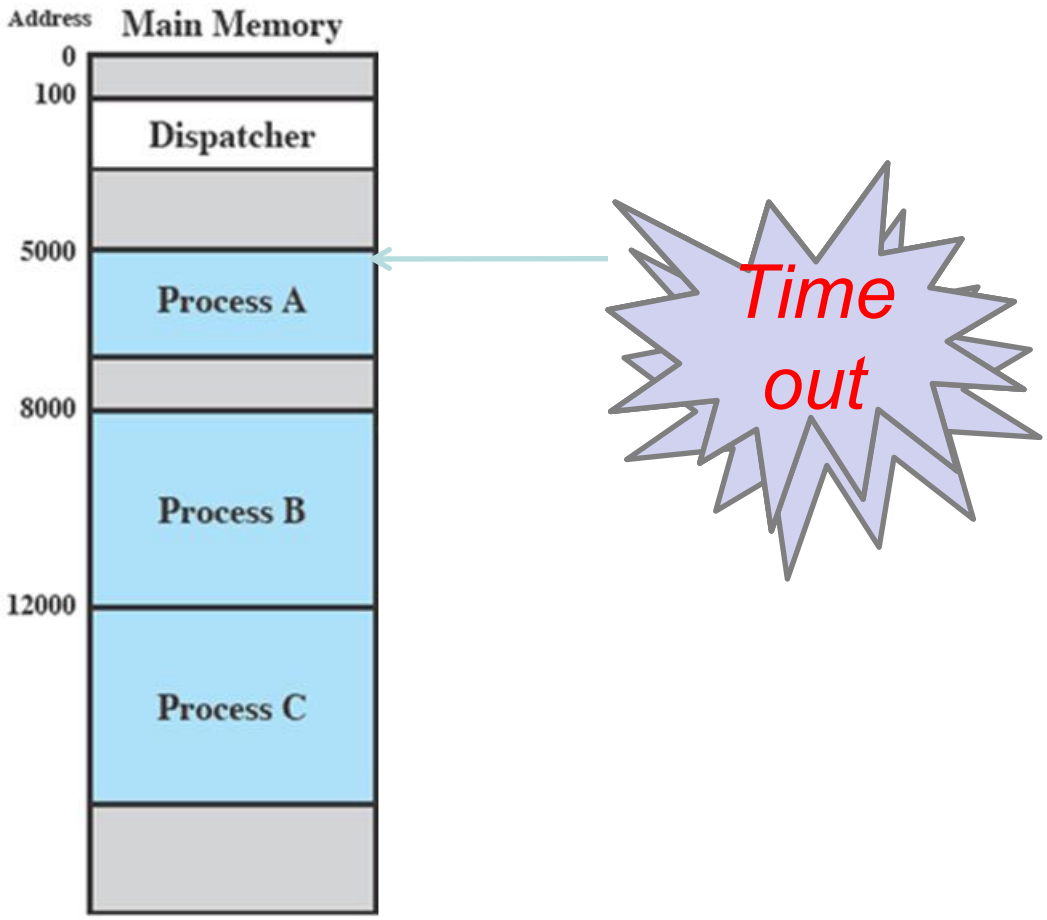


(a) Trace of Process A	(b) Trace of Process B	(c) Trace of Process C
5000	8000	12000
5001	8001	12001
5002	8002	12002
5003	8003	12003
5004		12004
5005		12005
5006		12006
5007		12007
5008		12008
5009		12009
5010		12010
5011		12011

5000 = Starting address of program of Process A
8000 = Starting address of program of Process B
12000 = Starting address of program of Process C

Figure 3.3 Traces of Processes of Figure 3.2

Trace from Processors point of view



Process States

- At any given time a process is either running or not running
- Number of states
  - Running
  - Not Running
- When the OS creates a process, the process is entered into which state?
  - Not Running

Five-state Process Model

- Running: currently being run
- Ready: ready to run
- Blocked: waiting for an event (I/O)
- New: just created, not yet admitted to set of run-able processes
- Exit: completed/error exit

Five-state Process Model

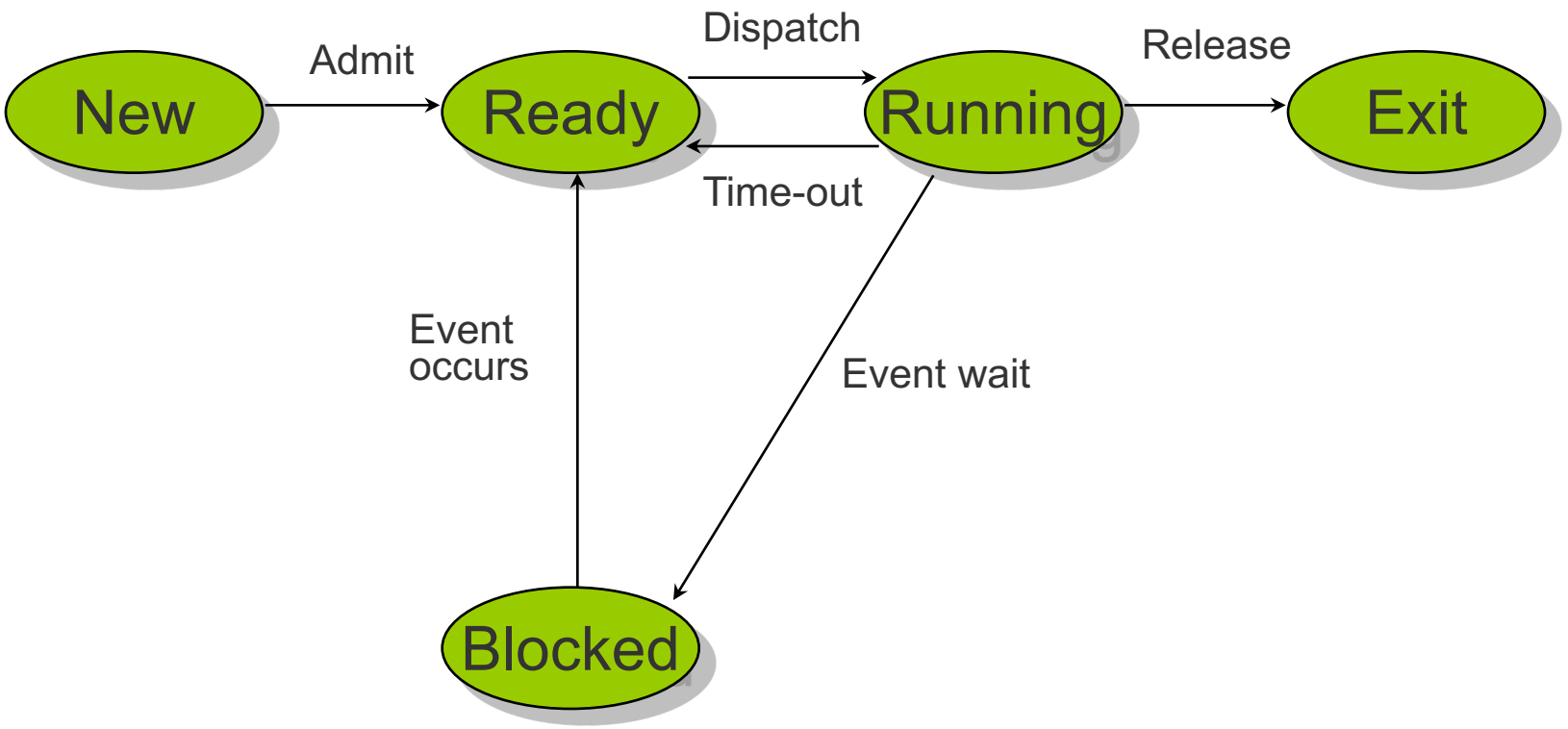
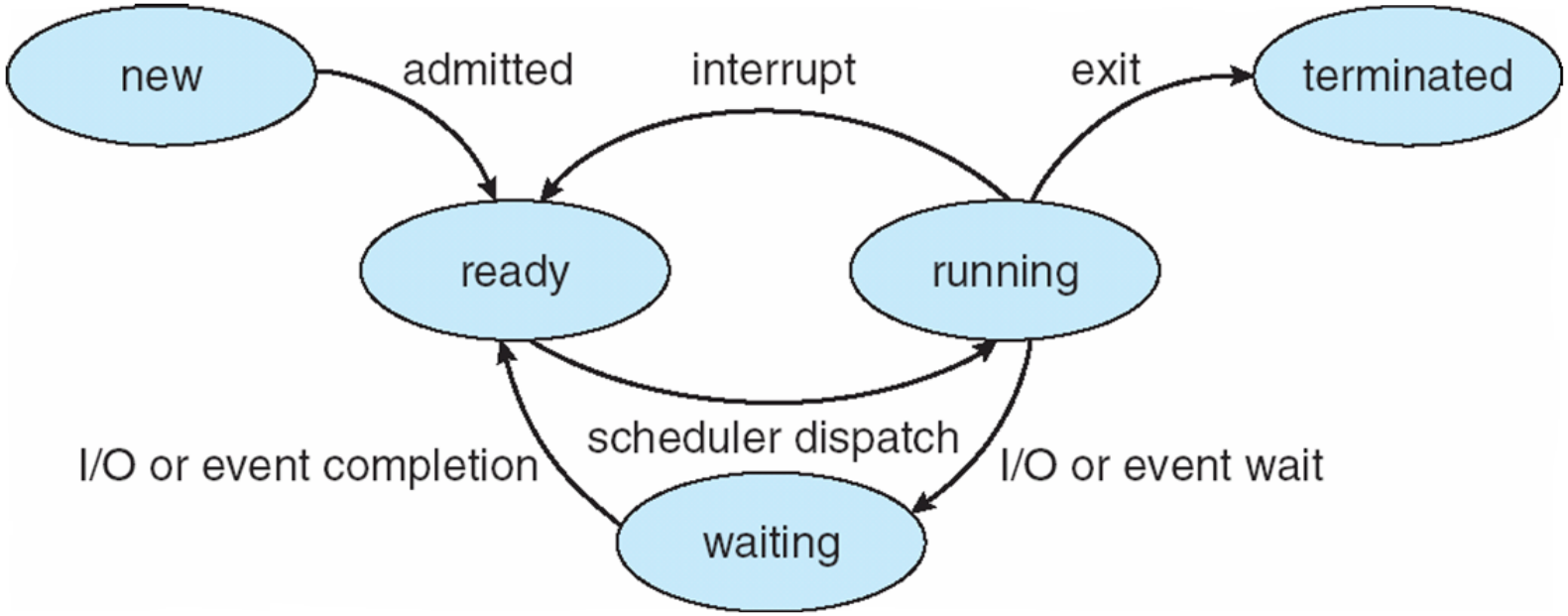
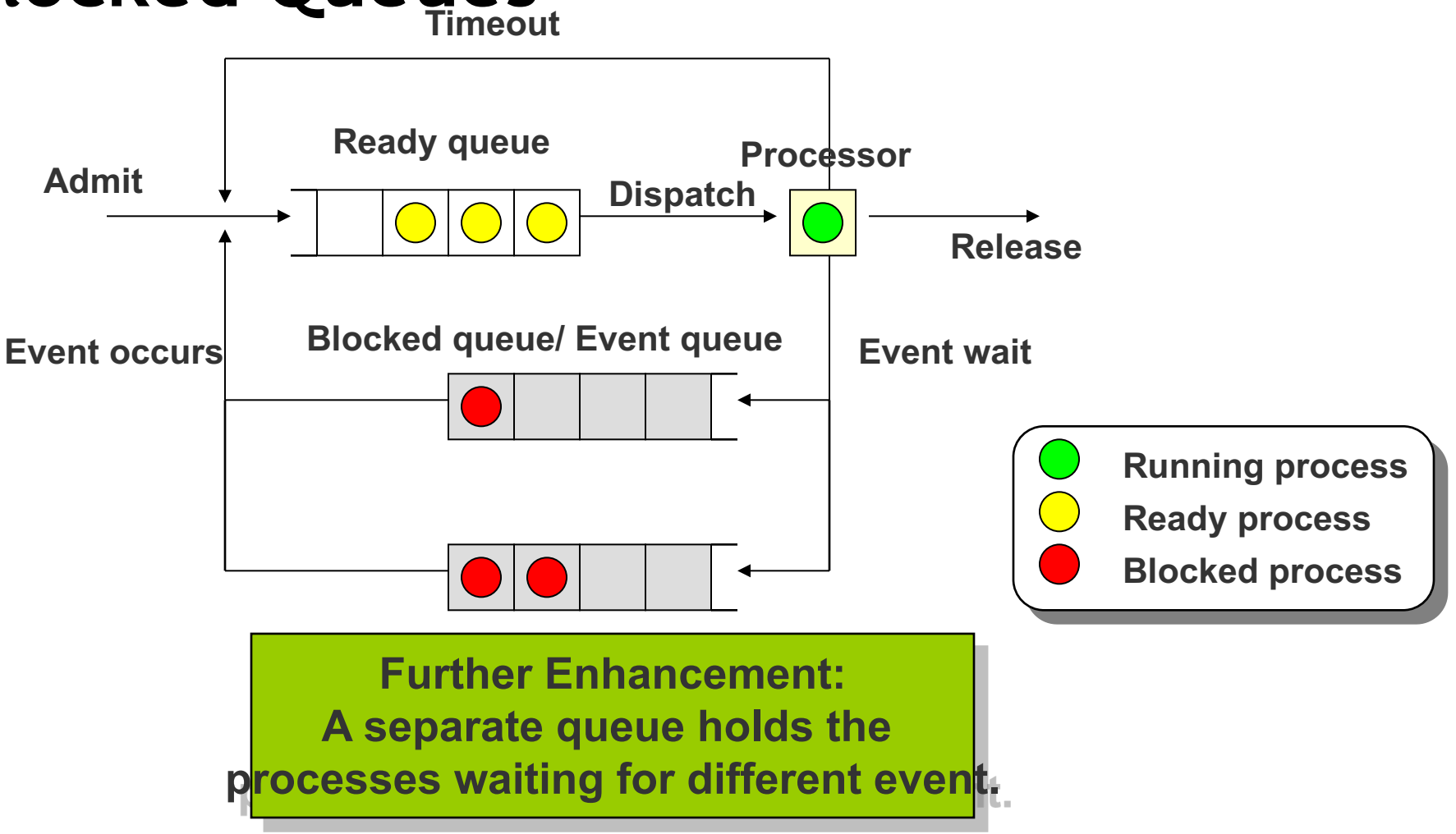


Diagram of Process State

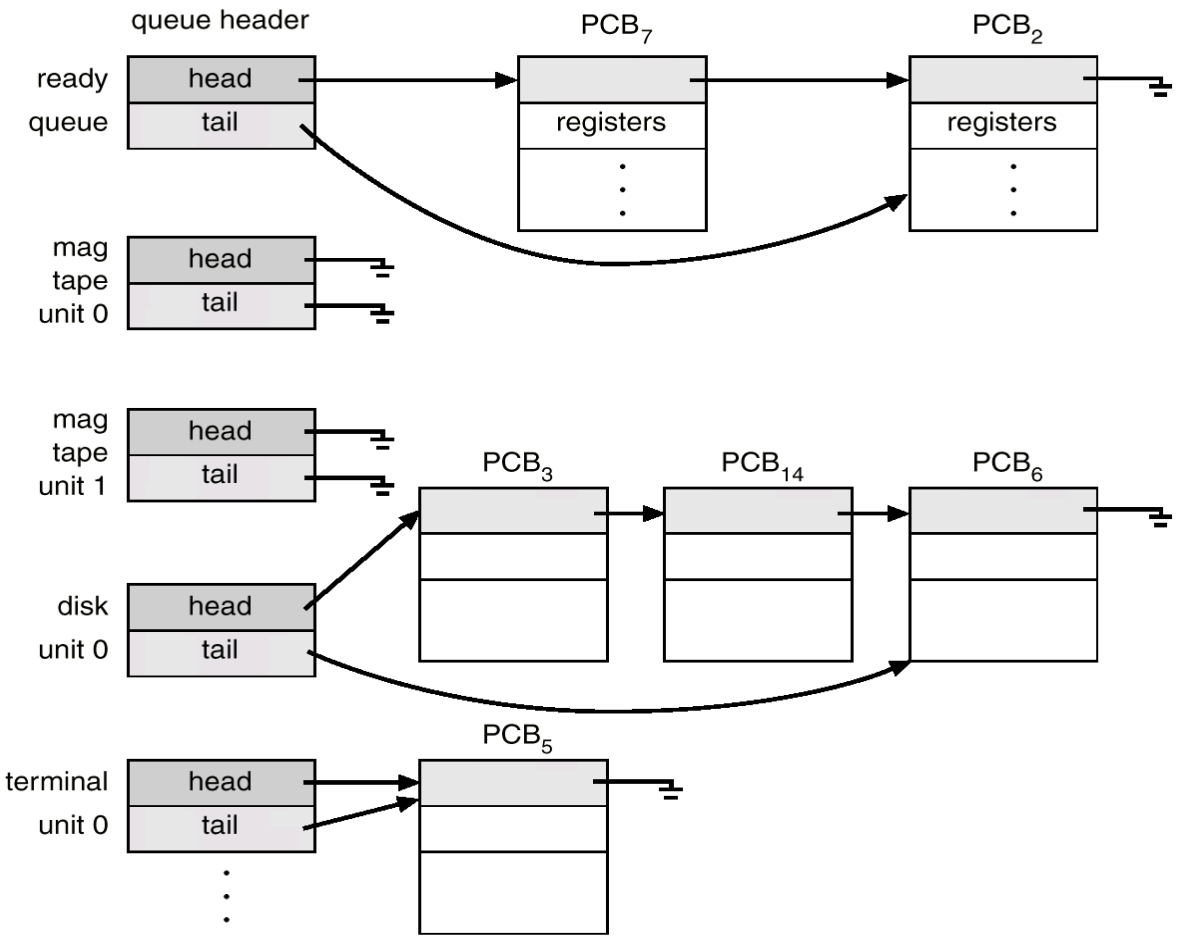


Blocked Queues



Further Enhancement:
A separate queue holds the processes waiting for different event.

Scheduling Queues



Schedulers

- Short term Scheduler or CPU Scheduling
  - Which program is to be run next
- Long term Scheduler or Job Scheduler
  - Which ready jobs should be brought to memory
  - May need to invoke only when a process leaves the system
  - Must make a careful selection

What's Happening at Process Creation?

- The OS:
  - Assigns a unique process identifier
  - Allocates space for the process
  - Initializes process control block
  - Creates or expand other data structures
- Traditionally, the OS created all processes
  - But it can be useful to let a running process create another
  - This action is called ***process spawning***
    - ***Parent Process*** is the original, creating, process
    - ***Child Process*** is the new process

OS Control Tables

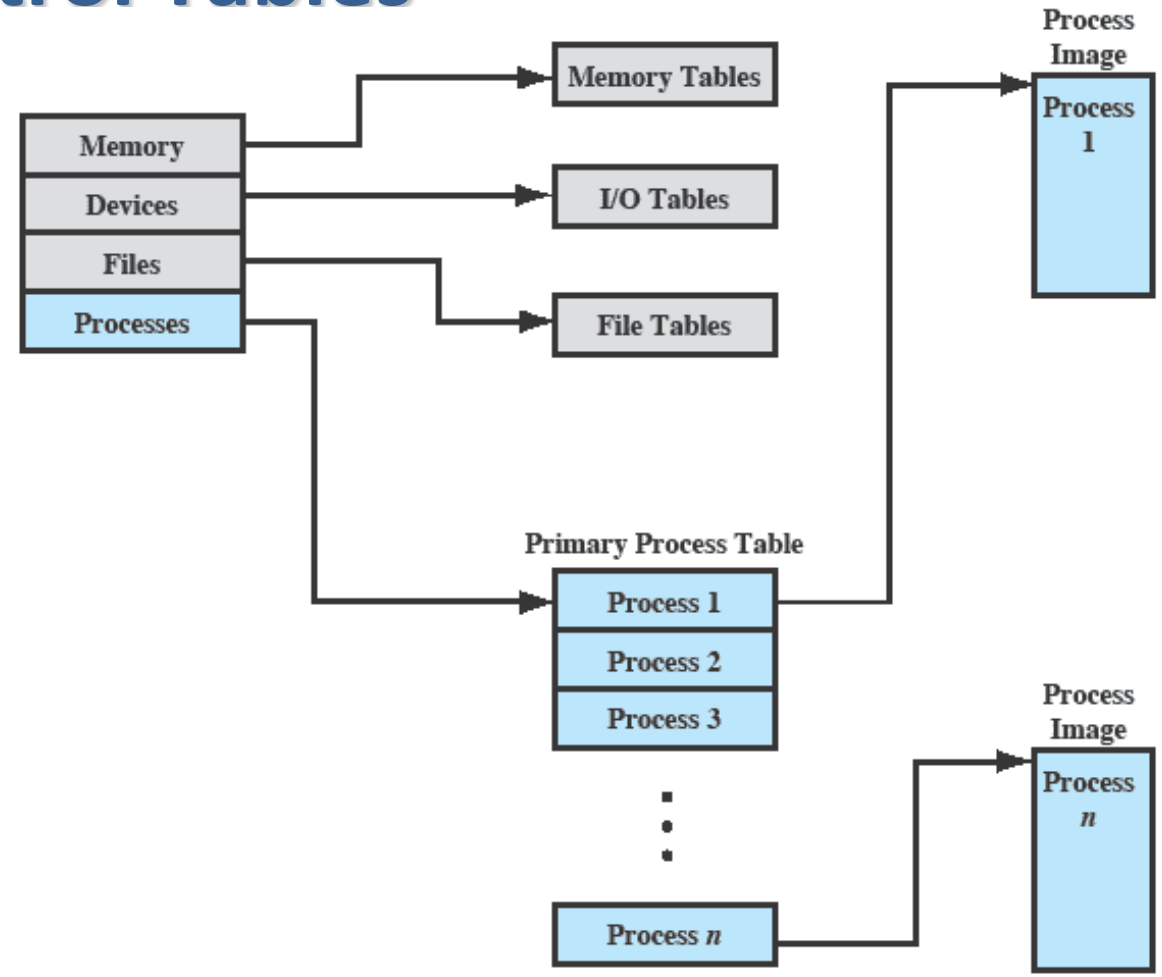


Figure 3.11 General Structure of Operating System Control Tables

Process Termination

- User logs off
- Process executes a service request to terminate
- Parent terminates so child processes terminate
- Operating system intervention
  - such as when deadlock occurs
- Error and fault conditions
  - E.g. memory unavailable, protection error, I/O failure, invalid instruction

Unix Process Creation

- When the system starts up it is running in kernel mode
- There is only one process, the kernel process.
- At the end of system initialization, the initial process starts up another kernel process.
  - The **init** kernel process has a process identifier of 1.

Process Creation

- These new processes may themselves go on to create new processes.
- All of the processes in the system are descended from the init kernel thread.
- You can see the family relationship between the running processes in a Linux system using the **pstree** command
- A new process is created by a **fork()** system call

Compiling C++ code

- g++ test.cpp –o Output
- Running the code:
- ./Output

The fork() system call

At the end of the system call there is a new process waiting to run once the scheduler chooses it

- A new data structure is allocated
- The new process is called the child process.
- The existing process is called the parent process.
- The parent gets the child's pid returned to it.
- The child gets 0 returned to it.
- Both parent and child execute at the same point after fork() returns

Unix Process Control

```
int main()
{
        int pid;
        int x = 0;

        x = x + 1;
        fork();
        x = 3;
        printf("%d",x);
}
```

But we want the child process to do something

```
int pid;
int status = 0;

pid = fork();
if (pid>0) {
      /* parent */
      ……
      pid = wait(&status);
} else {
      /* child */
      ……
      exit(status);
}
```

*The **fork** syscall returns a zero to the child and the child process ID to the parent*

*Parent uses **wait** to slee… chil… retu… and…*

***Fork** creates an exact copy of the parent process*

***Wait** variants allow wait on a specific child or*

*Child process passes status back to parent on **exit**, to report success/failure*

Child Process Inherits

- Stack
- Memory
- Environment
- Open file descriptors.
- Current working directory
- Resource limits
- Root directory

Child process DOESNOT Inherit

- Process ID
- Different parent process ID
- Process times
- Own copy of files
- Resource utilization (initialized to zero)

```
#include <iostream>
#include <unistd.h>

using namespace std;

int main()
{
int pid;
string x;


fork();
x = "hello";
cout << "\n" << x;
}
```

```
#include <iostream>
#include <unistd.h>


 using namespace std;

 int main()
 {
   int status;
 int pid = fork();
   if (pid == 0) {
       cout << "Hello - from child having id " <<
getpid() << " having parent " << getppid() << endl;

   }
   else {
        cout << "World - from Parent having id " <<
getpid() << endl;
   }
 }
```

How can a parent and child process communicate?

- Through any of the normal IPC mechanism schemes.
- But have special ways to communicate
  - For example
    - The variables are replicas
    - The parent receives the exit status of the child

The wait() System Call

- A child program returns a value to the parent, so the parent must arrange to receive that value
- The wait() system call serves this purpose
  - pid_t wait(int *status)
  - it puts the parent to sleep waiting for a child's result
  - when a child calls exit(), the OS unblocks the parent and returns the value passed by exit() as a result of the wait call (along with the pid of the child)
  - if there are no children alive, wait() returns immediately
  - also, if there are zombies, wait() returns one of the values immediately (and deallocates the zombie)

What is a zombie?

- In the interval between the child terminating and the parent calling wait(), the child is said to be a ‘zombie’.
- Even though its not running its taking up an entry in the process table.
- The process table has a limited number of entries.

What is a zombie?

- If the parent terminates without calling wait(), the child is adopted by init.

The solution is:

- Ensure that your parent process calls wait() or waitpid or etc, for every child process that terminates.

exit()

void exit(int *status*);

- After the program finishes execution, it calls *exit*()
- This system call:
  - takes the “result” of the program as an argument
  - closes all open files, connections, etc.
  - deallocates memory
  - deallocates most of the OS structures supporting the process
  - checks if parent is alive:
    - If so, it holds the result value until parent requests it, process does not really die, but it enters the zombie/defunct state
    - If not, it deallocates all data structures, the process is dead

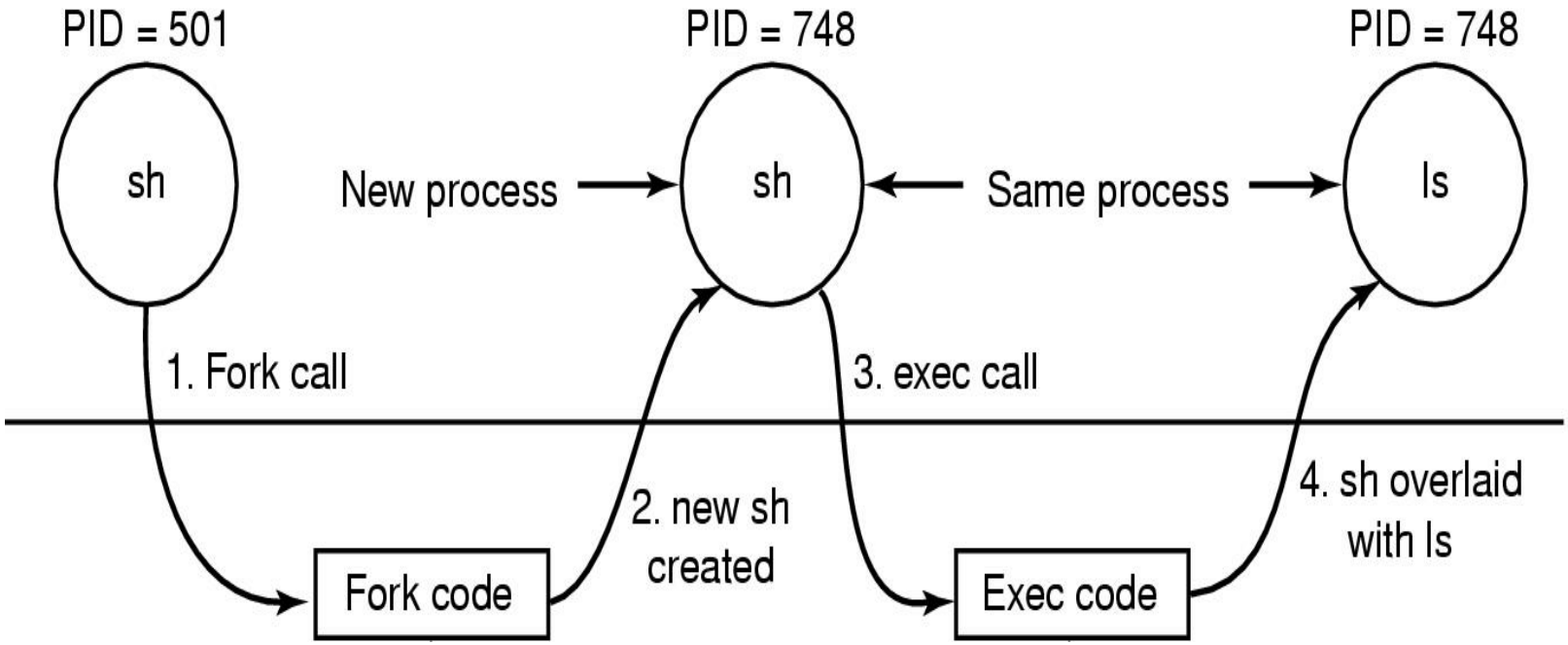
```
#include <iostream>
#include <unistd.h>
using namespace std;


int main()
{
  int status;
int pid = fork();
  if (pid == 0) {
       cout << "Hello - from child having id " << getpid()
<< " having parent " << getppid() << endl;
       exit(2);
  }
  else {
        cout << "World - from Parent having id " <<
getpid() << endl;
        wait(&status);
        cout << WEXITSTATUS(status) << endl;
  }
}
```

exec()

- We usually want the child process to run some other executable
- For Example, *ls - list directory contents*
- Overlays, performed by the family of exec system calls, can change the executing program, but can not create new processes.

The *ls* Command



Steps in executing the command *ls* type to the shell

execv

- int execv(const char *path, char *const argv[]);
- the current process image with a new process image.
- **path** is the filename to be executed by the child process
- When a C-language program is executed as a result of this call, it is entered as a C-language function call as follows:
  - int main (int argc, char *argv[]);
- The **argv** array is terminated by a null pointer.
- The null pointer terminating the **argv** array is not counted in **argc**.

```
#include <iostream>
#include <unistd.h>

using namespace std;

int main()
{
  int status;
int pid = fork();
  if (pid > 0) {
    cout << "World - from Parent having id " << getpid()
<< endl;
  }
  else {
    execv("./helloAOS", NULL);

  }
```

```
//helloAOS.cpp

#include <iostream>
using namespace std;

int main()
{
  cout << "Hello AOS
\n";
  return 0;
}
```

Assignment-01

- Will be announced in the next class.