IN1011

Operating Systems

Lecture 03 :
(part 1) Recap of What happens When an Interrupt Occurs
(part 2) Interleaving Processes Demo (fork(), execv(), wait())
(part 3) CPU Scheduling
(part 4) CPU Scheduling Criteria

IN1011

Operating Systems

Lecture 03 (part 1) :
Recap of What happens When an Interrupt Occurs

Recall Session 01: CPU Registers

Program counter (PC) stores the location in memory of the next instruction to be executed by the CPU.

Also important, but not shown, is the **program status word** (PSW) which defines status of the CPU (e.g. the current execution mode, status of latest logical and arithmetic operations). Some architectures combine the PC and PSW

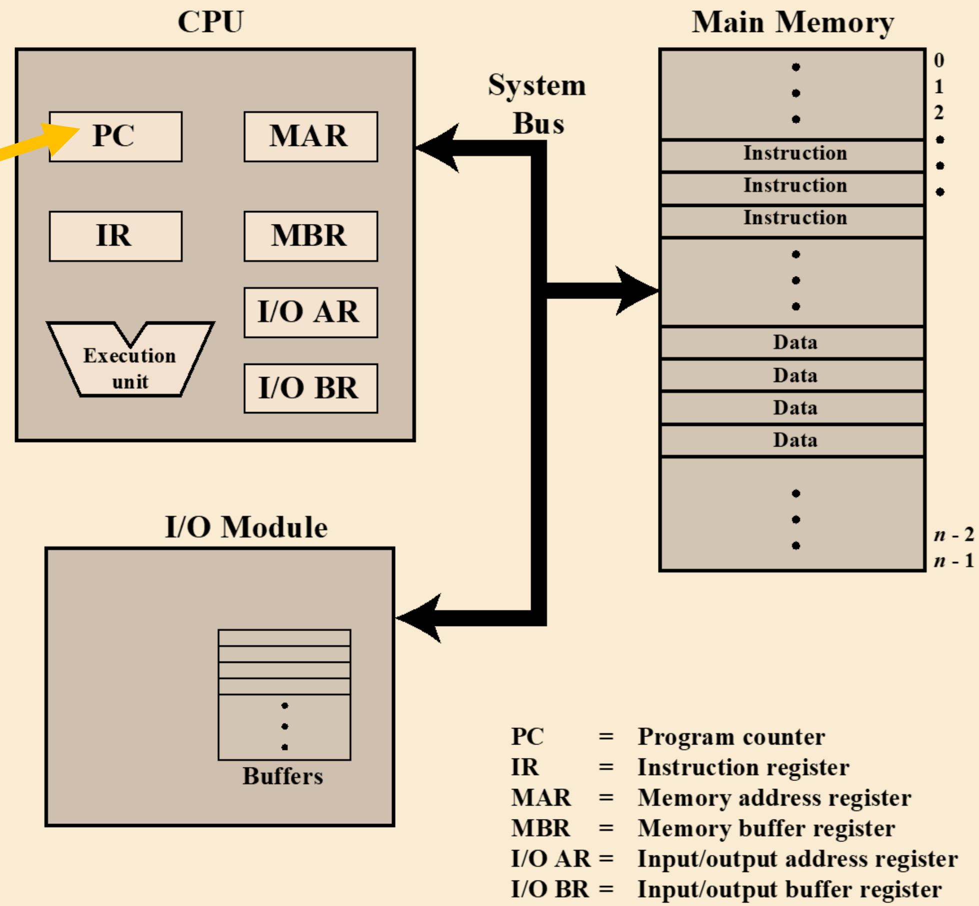


Figure 1.1 Computer Components: Top-Level View

Recall Session 01: Interrupts

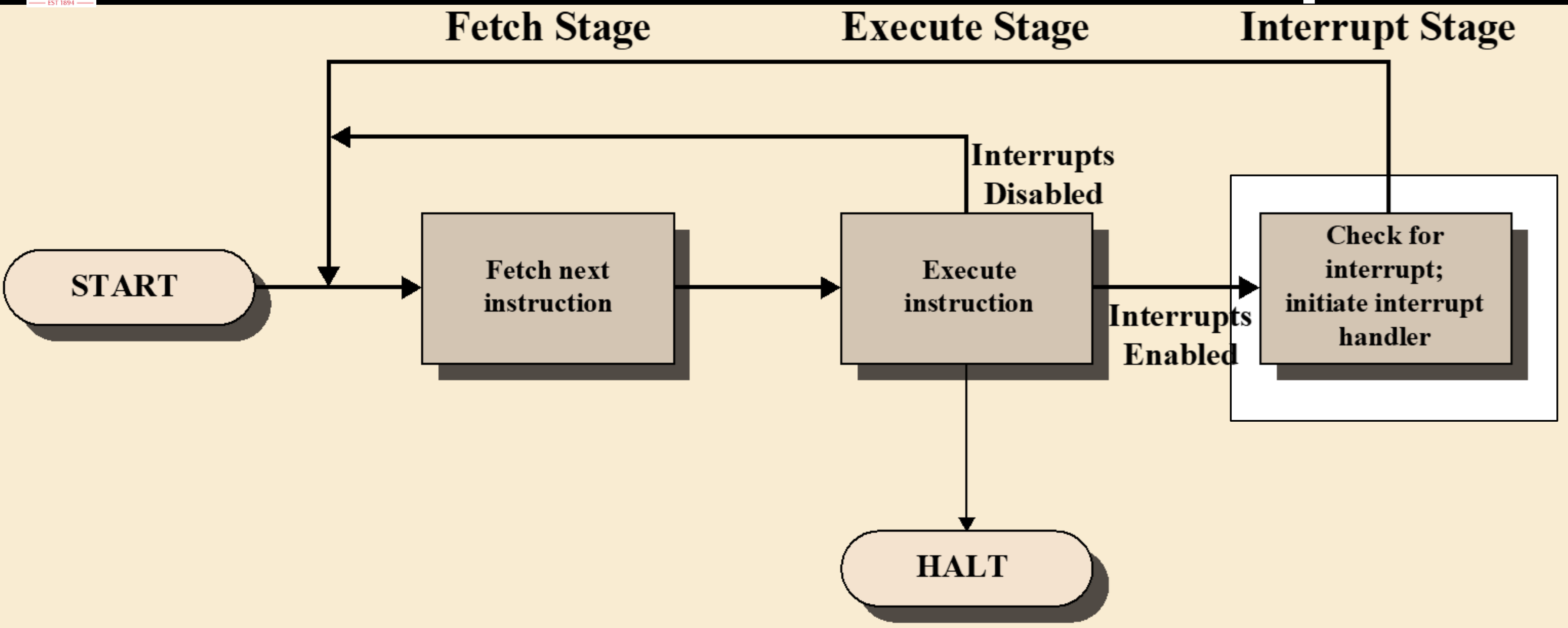
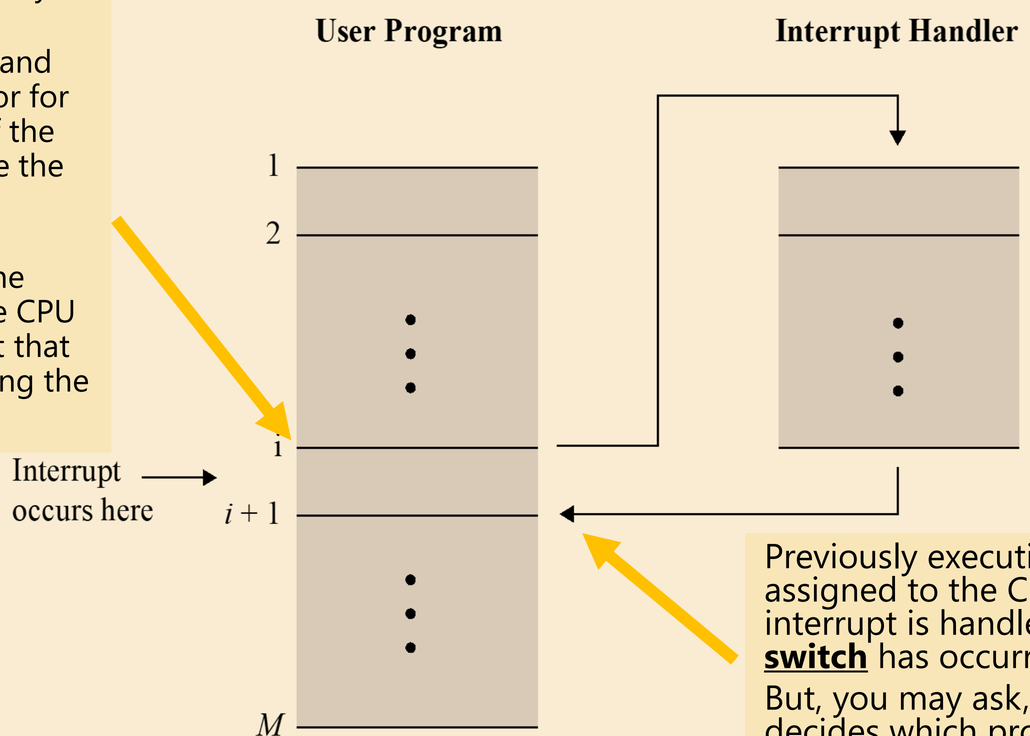


Figure 1.7 Instruction Cycle with Interrupts

Recall Sessions 01/02: Interrupts

<u>Mode switch</u> happens here.

- The CPU saves the contents of the PC and PSW registers in the control stack for the currently executing process;
- CPU enters **Kernel mode** and consults an interrupt vector for the location in memory of the OS routine that will handle the particular interrupt – an **interrupt handler**;
- The location is stored in the **program counter** and the CPU executes the instruction at that location next, thereby giving the OS control of the CPU;



Previously executing process is re-assigned to the CPU after the interrupt is handled. So no **<u>process switch</u>** has occurred here this time.

But, you may ask, which OS routine decides which process gets the CPU?

See later in this lecture for the answer.

Figure 1.6 Transfer of Control via Interrupts

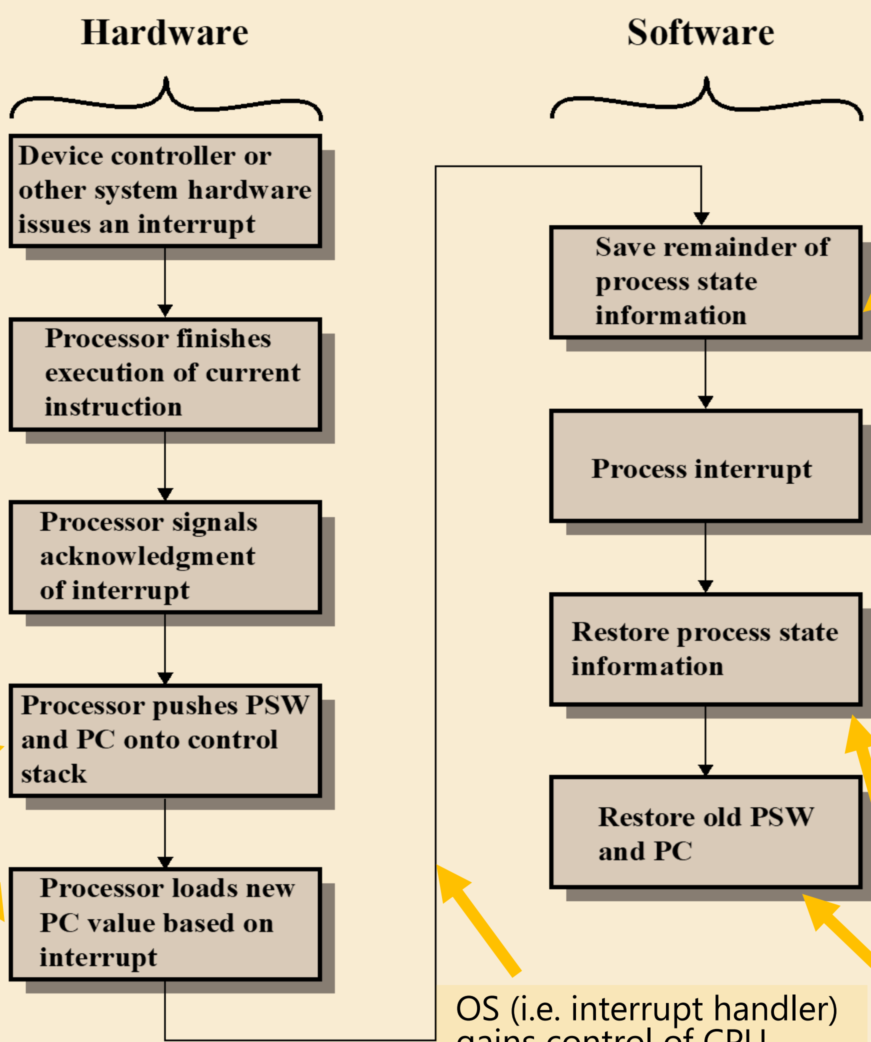


Figure 1.10 Simple Interrupt Processing

Mode switch

OS (i.e. interrupt handler) gains control of CPU, once mode switch has put CPU in kernel mode

The **process control block** (PCB) of the interrupted process is updated. For instance:

- Values of CPU registers are saved;
- State of the process is changed to ready;
- Scheduling information is saved, such as how long the process has been executing for;
- Pointers from/to relevant OS control structures are updated
- Information about open files;
- Memory management information.

Links with other OS control structures (e.g. other PCBs) are updated

After interrupt is handled, the previously running process can be reassigned to the CPU, using the saved information in its PCB. Also, the PC and PSW registers can be restored from interrupted process's control stack

10

IN1011

Operating Systems

Lecture 03 (part 2) : Interleaving Processes Demo fork(), execv(), wait()

IN1011

Operating Systems

Lecture 03 (part 2) : Interleaving Processes Demo fork(), execv(), wait()

Some Unix System Calls : fork(), execv() and wait()

- **fork()**:
  - when called, creates a copy of the process that makes the call.
  - the process ID of the "child" process is different from that of the "parent";
  - by default, the two processes have different process images (i.e. assigned to different parts of main memory);
  - both processes continue to execute from the point in the code after the call was made;
  - fork() returns the ID of the child process to the parent process. And, returns 0 to the child process;
- **execv()**:
  - when called, replaces the calling process with a new process executing a possibly new program;
  - The process ID remains unchanged;
- **wait()**:
  - default behavior, when called, is to suspend the calling process, until all child processes terminate, before continuing execution

fork(), execv(), wait(): Example Code 1

```cpp
#include <iostream>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

using namespace std;

void declare_process(pid_t & ref_pid, int & ref_numofproc)
{
    ref_numofproc*=2;

    if(ref_pid>0) //this is executed in the parent process, because the id of the child is returned to the parent by "fork"
    {
    cout << "There are "<< ref_numofproc << " processes at this level, according to process "<< getpid() << "\n" << flush;
    }else //this is executed in the child process, because "fork" call returns id 0 to the child process
    {

    cout << "There are "<< ref_numofproc << " processes at this level, according to process "<< getpid() << " and its parent "<< getppid() << "\n"<< flush;
    }
}

int main()
{
    int num_of_processes = 1;
    int num_of_fork_levels = 2;
    int status = 0; //used by "wait" call to check child statuses
    pid_t pid, wpid; //used to store return values from system calls "fork" and "wait"


    while(num_of_fork_levels>0)
    {
        num_of_fork_levels--; //reduce the number of fork levels by 1

        pid = fork();
        if(-1==pid) //no child was created
        {
        perror("fork");
        exit(EXIT_FAILURE);
        }

        declare_process(pid,num_of_processes);
    }

    while((wpid=wait(&status))>0);

    if(pid==0) //this is a child process
    {
        char * ls_args[] = { "/bin/ps", "-f", "--forest", NULL};

        //execute the program
         execv(   ls_args[0],     ls_args);
    }


    return 0;
}
```

"fork()" is called to create copy of running process

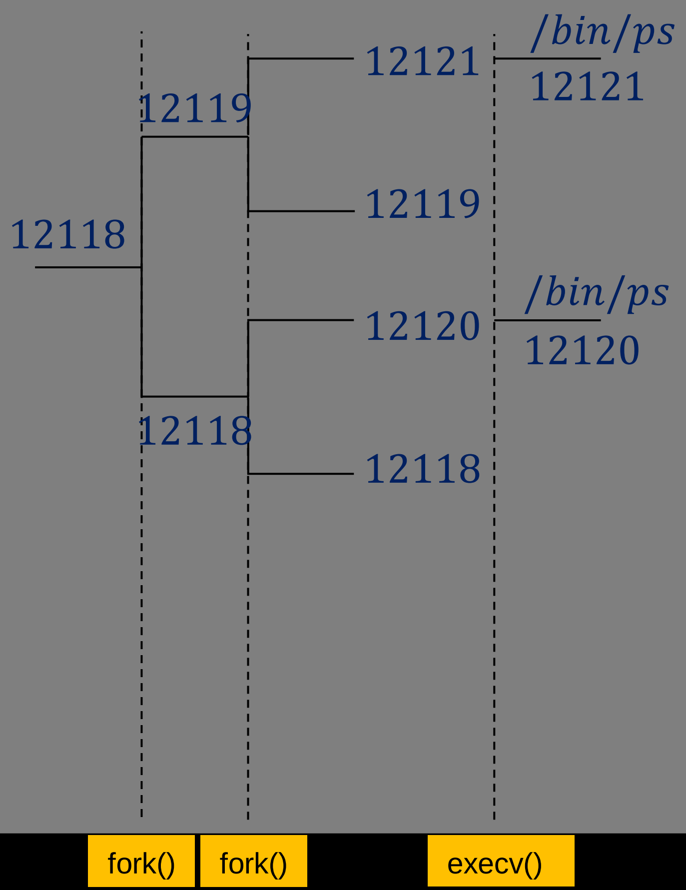
This call to "wait()" suspends the parent process till all of its children have terminated. **<u>What is the consequence of calling wait at this point in the code?</u>**

remember the "ps" command from last week's tutorial. When called in a Unix bash shell, it prints out (to the standard output) information for all currently running processes.

This line in the code is preparing the command as a "C"-string, so that we can call the command using "execv()".

"execv()" is called to turn running process into a process that executes "ps" unix command

Order of Interleaved Processes:Example Code 1



This follows a "breadth-first" execution of the processes!!

```
There are 2 processes at this level, according to process 12118
There are 2 processes at this level, according to process 12119 and its parent 12118
There are 4 processes at this level, according to process 12118
There are 4 processes at this level, according to process 12119
There are 4 processes at this level, according to process 12121 and its parent 12119
There are 4 processes at this level, according to process 12120 and its parent 12118
UID        PID  PPID  C STIME TTY          TIME CMD
runner8  12113     0  0 21:18 pts/1    00:00:00 sh -c ./a.out
runner8  12118 12113  0 21:18 pts/1    00:00:00  \_ ./a.out
runner8  12119 12118  0 21:18 pts/1    00:00:00      \_ ./a.out
runner8  12121 12119  0 21:18 pts/1    00:00:00      |   \_ /bin/ps -f --forest
runner8  12120 12118  0 21:18 pts/1    00:00:00      \_ ./a.out
UID        PID  PPID  C STIME TTY          TIME CMD
runner8  12113     0  0 21:18 pts/1    00:00:00 sh -c ./a.out
runner8  12118 12113  0 21:18 pts/1    00:00:00  \_ ./a.out
runner8  12119 12118  0 21:18 pts/1    00:00:00      \_ [a.out] <defunct>
runner8  12120 12118  0 21:18 pts/1    00:00:00      \_ /bin/ps -f --forest
```

Calling fork(), execv() and wait()

```cpp
#include <iostream>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

using namespace std;

void declare_process(pid_t & ref_pid, int & ref_numofproc)
{
    ref_numofproc*=2;

    if(ref_pid>0) //this is executed in the parent process, because the id of the child is returned to the parent by "fork"
    {
    cout << "There are "<< ref_numofproc << " processes at this level, according to process "<< getpid() << "\n" << flush;
    }else //this is executed in the child process, because "fork" call returns id 0 to the child process
    {

    cout << "There are "<< ref_numofproc << " processes at this level, according to process "<< getpid() << " and its parent "<< getppid() << "\n"<< flush;
    }
}

int main()
{
    int num_of_processes = 1;
    int num_of_fork_levels = 2;
    int status = 0; //used by "wait" call to check child statuses
    pid_t pid, wpid; //used to store return values from system calls "fork" and "wait"


    while(num_of_fork_levels>0)
    {
        num_of_fork_levels--; //reduce the number of fork levels by 1

        pid = fork();
        if(-1==pid) //no child was created
        {
        perror("fork");
        exit(EXIT_FAILURE);
        }

        declare_process(pid,num_of_processes);
    }

    while((wpid=wait(&status))>0);

    if(pid==0) //this is a child process
    {
        char * ls_args[] = { "/bin/ps", "-f", "--forest", NULL};

        //execute the program
          execv(   ls_args[0],     ls_args);
    }



    return 0;
}

```

Calling fork(), execv() and wait()

```
#include <iostream>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

using namespace std;

void declare_process(pid_t & ref_pid, int & ref_numofproc)
{
    ref_numofproc*=2;

    if(ref_pid>0) //this is executed in the parent process, because the id of the child is returned to the parent by "fork"
    {
        cout << "There are "<< ref_numofproc << " processes at this level, according to process "<< getpid() << "\n" << flush;
    }else //this is executed in the child process, because "fork" call returns id 0 to the child process
    {
        cout << "There are "<< ref_numofproc << " processes at this level, according to process "<< getpid() << " and its parent "<< getppid() << "\n"<< flush;
    }
}

int main()
{
    int num_of_processes = 1;
    int num_of_fork_levels = 2;
    int status = 0; //used by "wait" call to check child statuses
    pid_t pid, wpid; //used to store return values from system calls "fork" and "wait"


    while(num_of_fork_levels>0)
    {
        num_of_fork_levels--; //reduce the number of fork levels by 1

        pid = fork();
        if(-1==pid) //no child was created
        {
            perror("fork");
            exit(EXIT_FAILURE);
        }

        if(pid>0){while((wpid=wait(&status))>0);} //if this is the parent process, do not proceed further untill all children have terminated

        declare_process(pid,num_of_processes);
    }

    if(pid==0) //this is a child process
    {
        char * ls_args[] = { "/bin/ps", "-f", "--forest", NULL};

        //execute the program
         execv(   ls_args[0],     ls_args);
    }



    return 0;
}
```

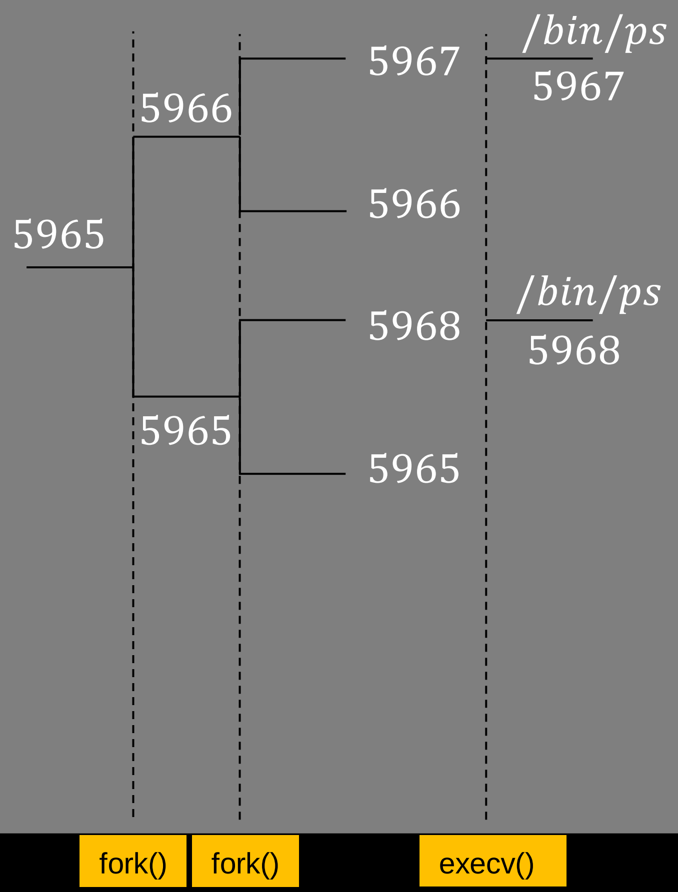
Same code as before, with “wait()” call in a different location. **<u>What is the consequence of calling wait at this point in the code?</u>**

Try this out using g++ in JSLinux (for compile command, see IN1011 Moodle page, Session 3)

```cpp
#include <iostream>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

using namespace std;

void declare_process(pid_t & ref_pid, int & ref_numofproc)
{
    ref_numofproc*=2;

    if(ref_pid>0) //this is executed in the parent process, because the id of the child is returned to the parent by "fork"
    {
        cout << "There are "<< ref_numofproc << " processes at this level, according to process "<< getpid() << "\n" << flush;
    }else //this is executed in the child process, because "fork" call returns id 0 to the child process
    {
        cout << "There are "<< ref_numofproc << " processes at this level, according to process "<< getpid() << " and its parent "<< getppid() << "\n"<< flush;
    }
}

int main()
{
    int num_of_processes = 1;
    int num_of_fork_levels = 2;
    int status = 0; //used by "wait" call to check child statuses
    pid_t pid, wpid; //used to store return values from system calls "fork" and "wait"


    while(num_of_fork_levels>0)
    {
        num_of_fork_levels--; //reduce the number of fork levels by 1

        pid = fork();
        if(-1==pid) //no child was created
        {
            perror("fork");
            exit(EXIT_FAILURE);
        }

        if(pid>0){while((wpid=wait(&status))>0);} //if this is the parent process, do not proceed further untill all children have terminated

        declare_process(pid,num_of_processes);
    }

    if(pid==0) //this is a child process
    {
        char * ls_args[] = { "/bin/ps", "-f", "--forest", NULL};

        //execute the program
        execv(   ls_args[0],     ls_args);
    }



    return 0;
}
```

Same code as before, with “wait()” call in a different location. **<u>What is the consequence of calling wait at this point in the code?</u>**

Order of Interleaved Processes:Example Code 2



This follows a "<u>depth-first</u>" execution of the processes!!

```
There are 2 processes at this level, according to process 5966 and its parent 5965
There are 4 processes at this level, according to process 5967 and its parent 5966
UID        PID  PPID  C STIME TTY          TIME CMD
14067     5960     0  0 21:25 pts/1    00:00:00 sh -c ./a.out
14067     5965  5960  0 21:25 pts/1    00:00:00  \_ ./a.out
14067     5966  5965  0 21:25 pts/1    00:00:00      \_ ./a.out
14067     5967  5966  0 21:25 pts/1    00:00:00          \_ /bin/ps -f --forest
There are 4 processes at this level, according to process 5966
There are 2 processes at this level, according to process 5965
There are 4 processes at this level, according to process 5968 and its parent 5965
UID        PID  PPID  C STIME TTY          TIME CMD
14067     5960     0  0 21:25 pts/1    00:00:00 sh -c ./a.out
14067     5965  5960  0 21:25 pts/1    00:00:00  \_ ./a.out
14067     5968  5965  0 21:25 pts/1    00:00:00      \_ /bin/ps -f --forest
There are 4 processes at this level, according to process 5965
```

IN1011

Operating Systems

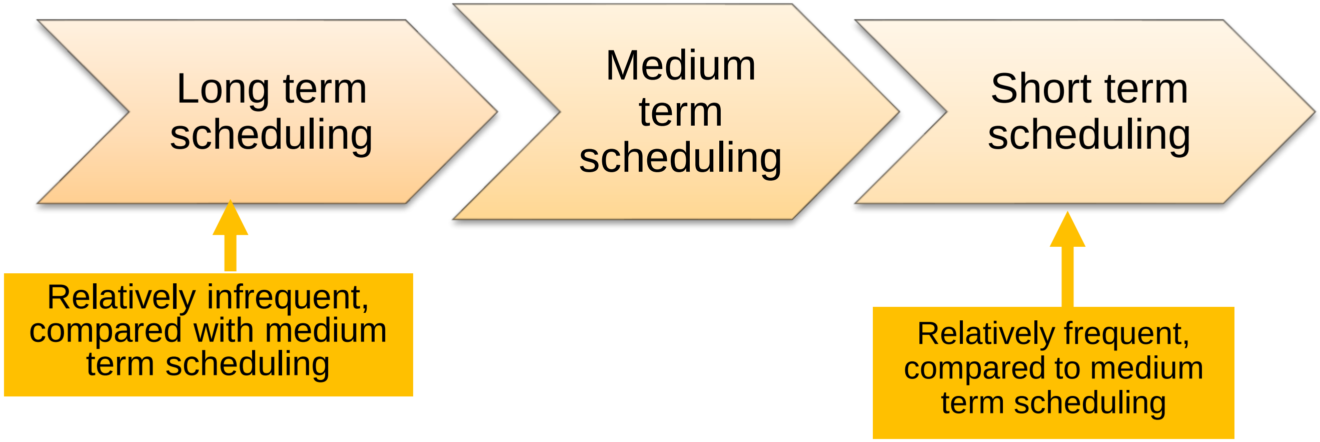
Lecture 03 (part 3):
CPU Scheduling

Question

- How does an Operating System decide when to change the state of processes?

CPU Scheduling

- Aim is to assign processes to be executed by the CPU in a way that meets system objectives
- 3 separate scheduling decisions

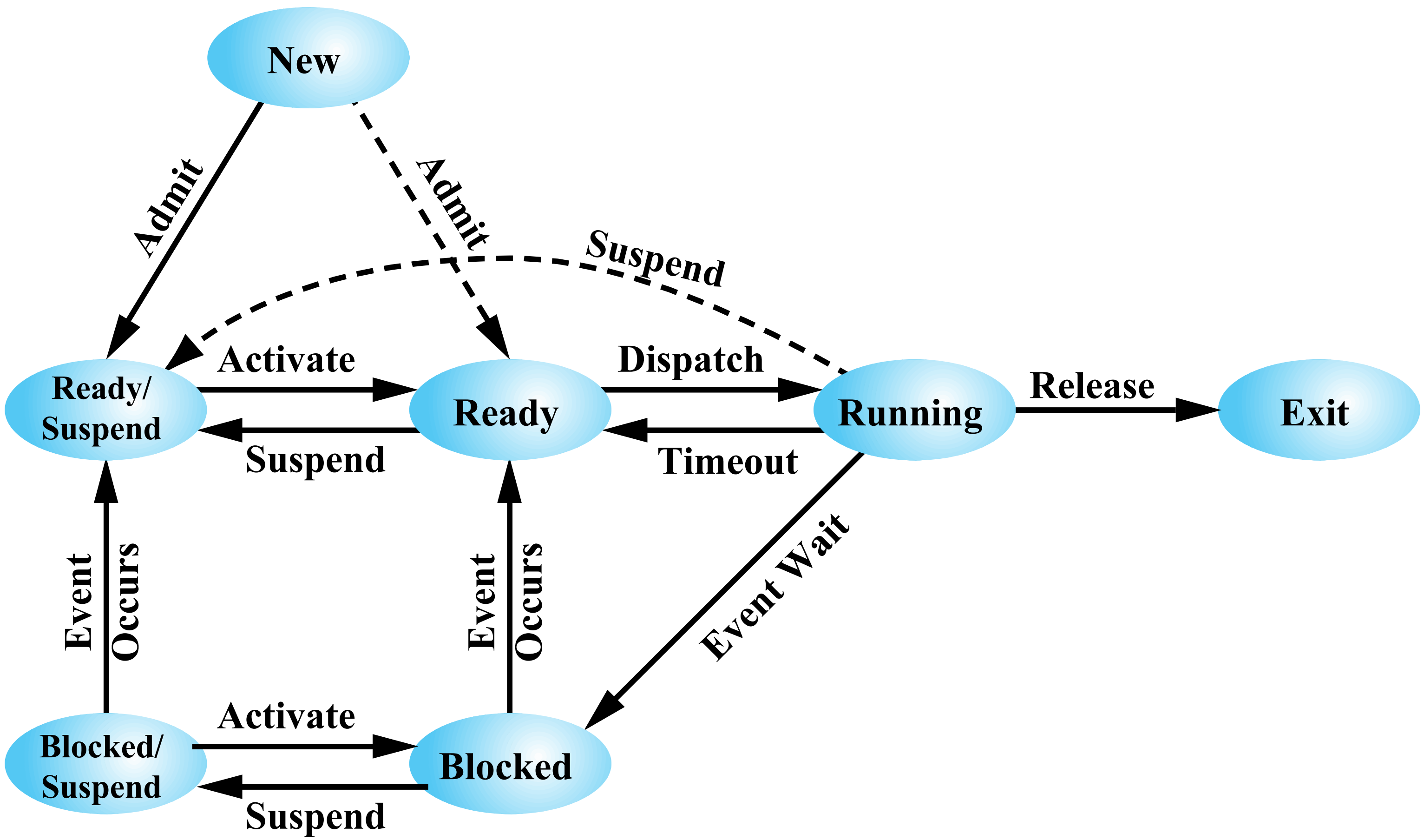


Types of Scheduling

Long-term scheduling	The decision to add to the pool of processes to be executed
Medium-term scheduling	The decision to add to the number of processes that are partially or fully in main memory
Short-term scheduling	The decision as to which available process will be executed by the processor
I/O scheduling	The decision as to which process's pending I/O request shall be handled by an available I/O device

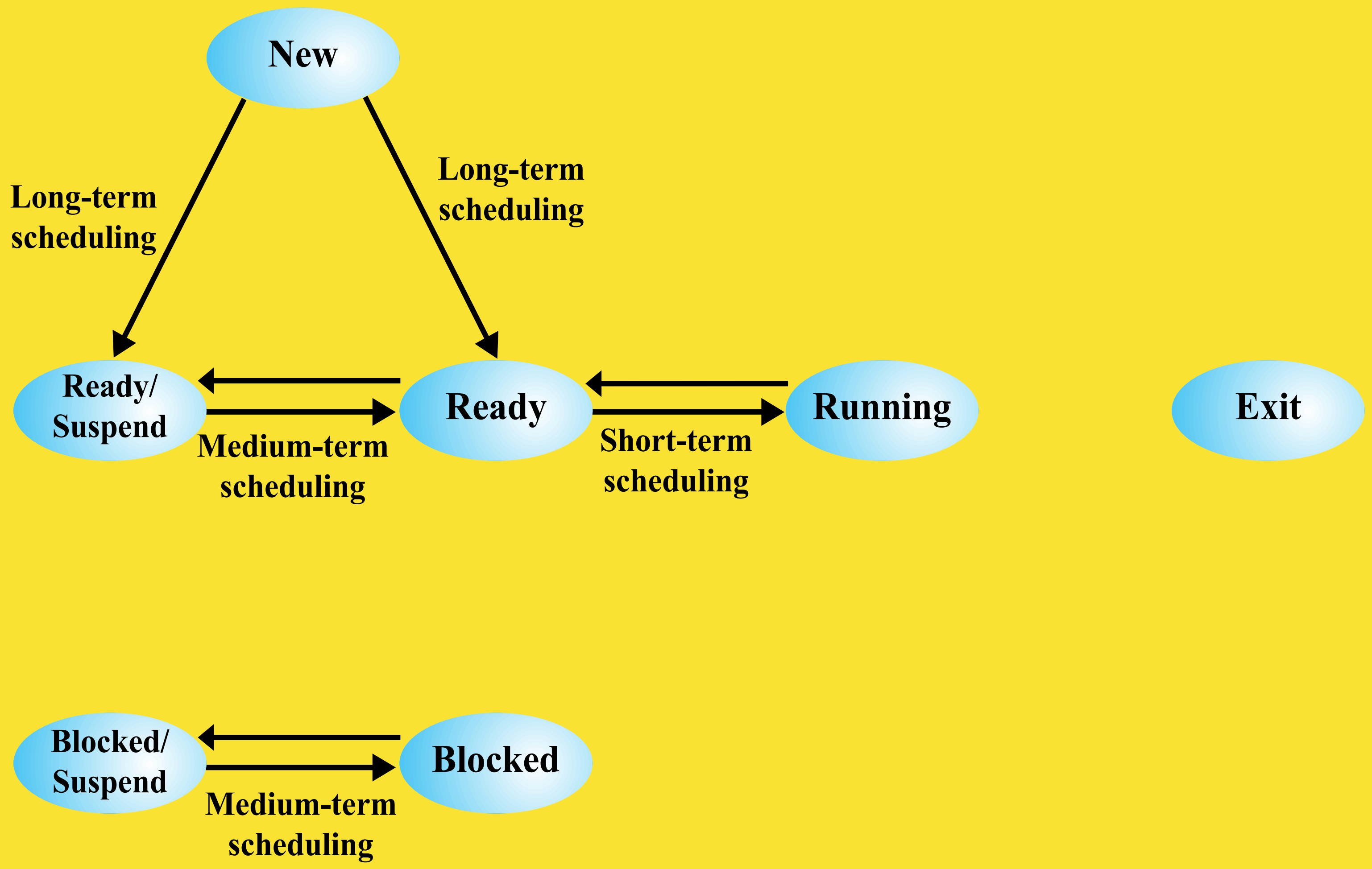
Table 9.1

CPU Scheduling (contd.)



(b) With Two Suspend States

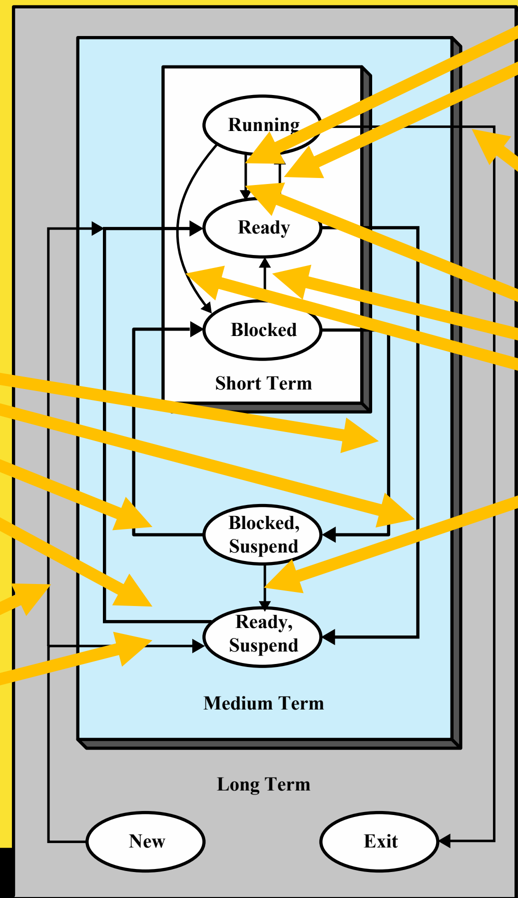
CPU Scheduling (contd.)



CPU Scheduling (contd.)

Short term scheduler determines which process is assigned to the CPU by the **Dispatcher**. The scheduler may be invoked when, for instance, an interrupt, trap or system call occurs. Or when a process terminates, or a process joins the ready queue.

Medium term scheduler determines the processes to be swapped into, or out of, secondary storage (e.g. hard disk).

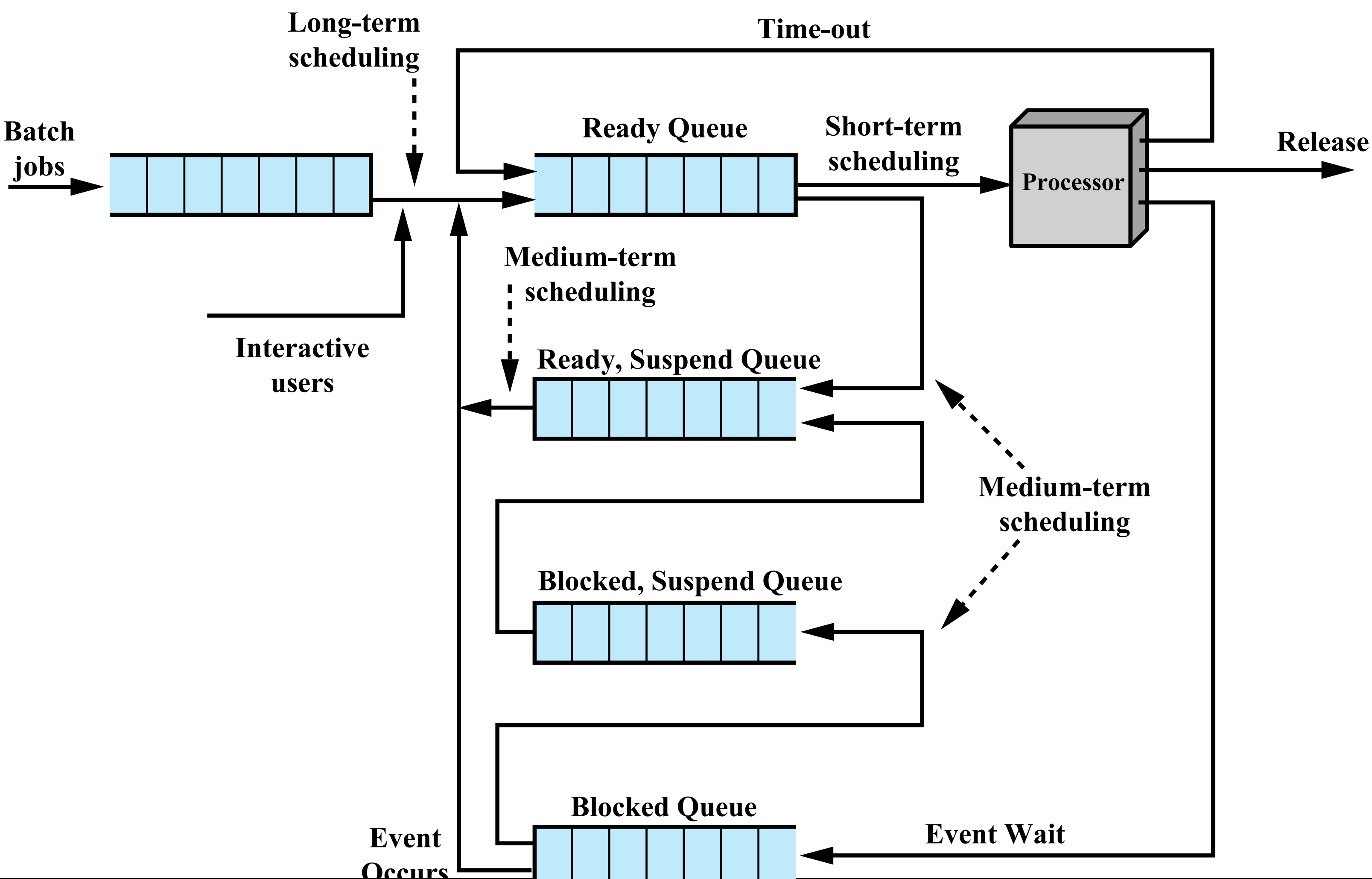


What are some of the causes of the remaining transitions, as these are not necessarily due to schedulers?

Ans: interrupts, traps, I/O requests, process termination, process priority, empty ready queue and non-empty ready/suspend queue

The **long term scheduler** admits the process into main memory, thereby putting the process into the ready or ready suspend state.

CPU Scheduling Queues



Long term Scheduler

- Determines which programs are admitted to the system for processing
- Controls the degree of multiprogramming
  - The more processes that are admitted, the smaller the percentage of time for each process to run;
  - Scheduler may limit the number of admitted processes to provide satisfactory service to the current set of processes;

- Creates processes from the new queue when it can, but must decide:
  - 1) When the operating system can take on one or more additional processes
  - 2) Which jobs to accept and turn into processes, based on one or more of the following:
    - first-come first-served, priority
    - expected execution time, I/O requirements

Medium term Scheduler

- Part of the OS's swapping function, moving a process out of main memory to secondary storage or vice-versa;
- "Swapping in/out" decisions are based on the need to manage the degree of multiprogramming (i.e. the mix of ***CPU bound*** and ***I/O bound*** processes)
  - CPU bound processes would go significantly faster if the CPU was faster;
  - I/O bound processes would go significantly faster if I/O access was faster;
  - Considers the memory requirements of the swapped-in processes. This information is stored in each process's **process control block** (PCB)

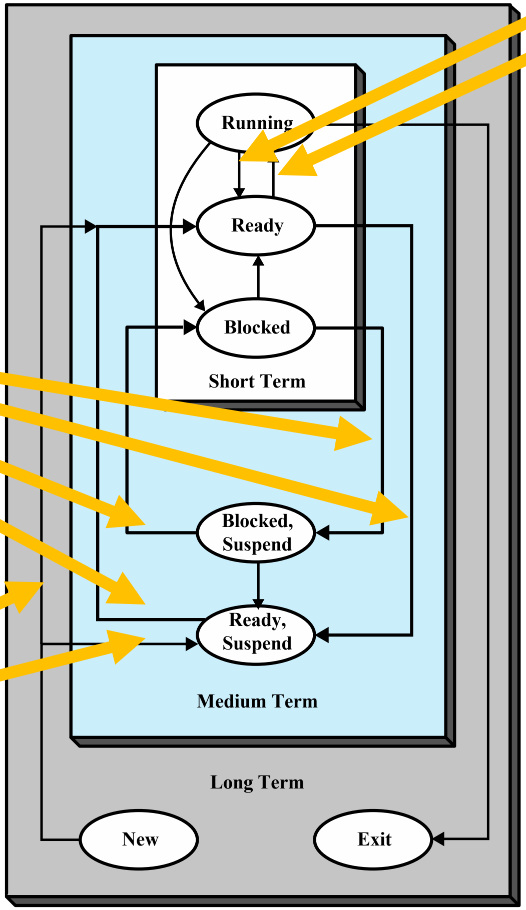
Short term Scheduler

- Decides which processes are assigned to the CPU by the **dispatcher**
- Executes most frequently
- Makes the fine-grained decisions of which process to execute next;
- Invoked when an event occurs that could result in the CPU being assigned to another process, e.g. when ***interrupts***, ***traps*** or certain ***system calls*** occur, when a process yields the CPU, or a higher priority process enters the ready queue;

Short term scheduler determines which process the CPU is assigned to, or released from, by the **Dispatcher**. The scheduler may be invoked when, for instance, an interrupt, trap or system call occurs. Or when a process terminates, or a process joins the ready queue.

Medium term scheduler determines the processes to be swapped into, or out of, secondary storage (e.g. hard disk).

The **long term scheduler** admits the process into main memory, thereby putting the process into the <u>ready</u> or <u>ready suspend</u> state.



IN1011

Operating Systems

Lecture 3 (part 4):
CPU Scheduling Criteria

Question (contd.)

- How does an Operating System decide when to change the state of processes?
  - special software modules called schedulers decide
- What criteria do scheduler's base their choices on?

Short term Scheduling Criteria

- Main objective is to allocate processor time in order to optimise certain aspects of system behavior
- A set of criteria is needed to evaluate the scheduling policy:

User-oriented criteria

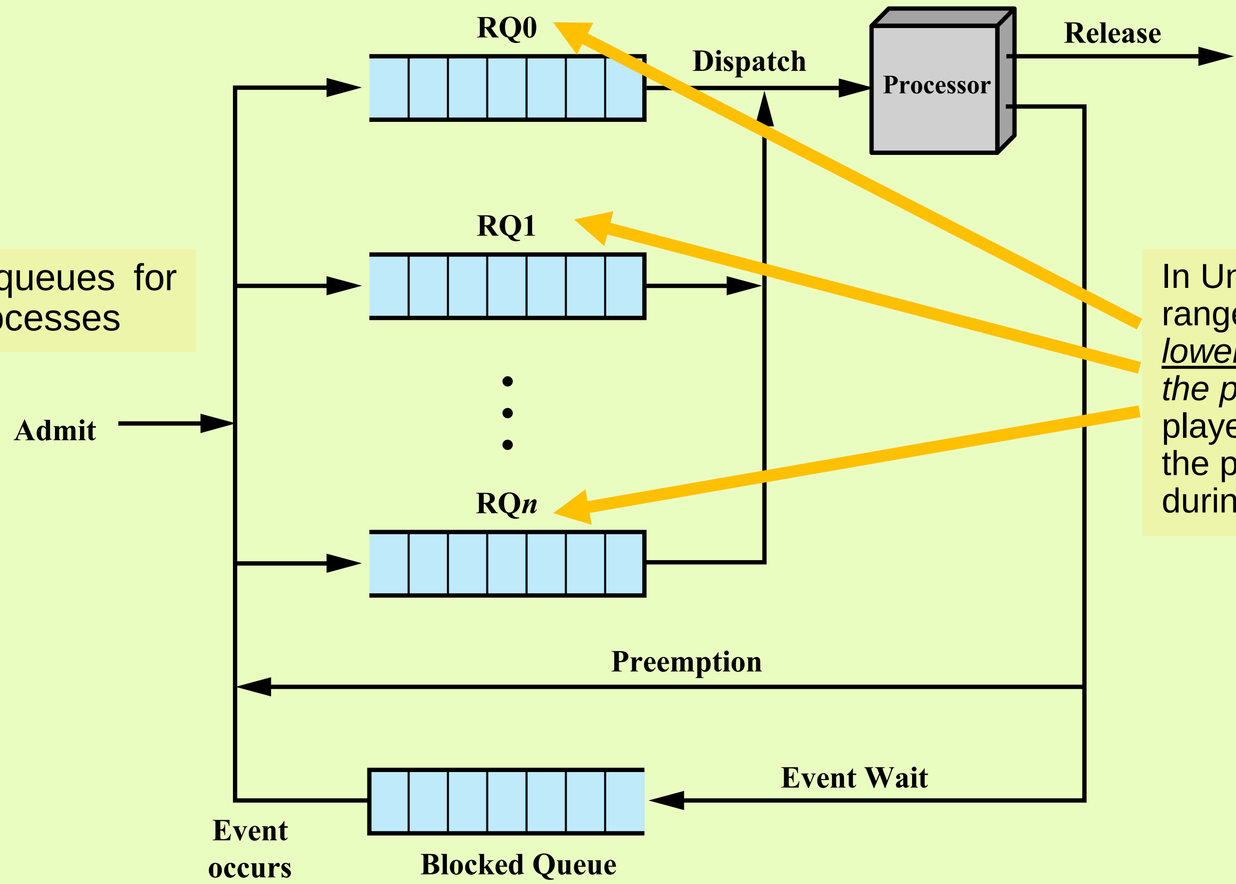
- Relate to the behavior of the system as perceived by the individual user or process
- Important for almost all systems

System-oriented criteria

- Focus is on effective and **efficient utilization** of the CPU
- Generally of minor importance on single-user systems

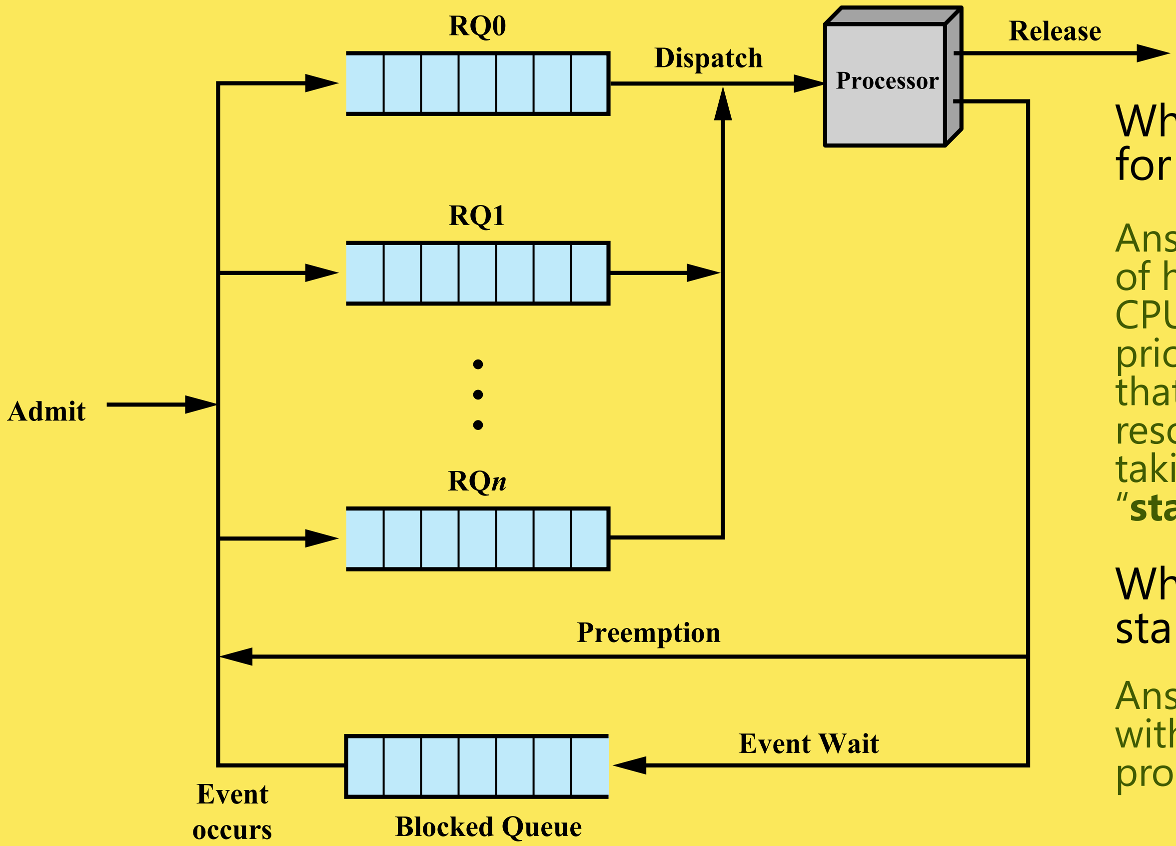
Short term Scheduler: Priority Queues

Multiple queues for ready processes



In Unix systems priorities range from -20 to 19. *The lower the value, the higher the priority.* You will have played around with changing the priorities of processes during your last lab

However,…



Why is this possibly a problem for low priority processes?

Ans: there may be a constant stream of higher priority processes that the CPU is assigned to, leaving lower priority processes unserved. Processes that are not assigned hardware resources due to other processes taking priority are said to experience "**starvation**".

What are some ways to prevent starvation?

Ans: priority can be made to change with age or execution history of a process

Scheduling Policies

- The scheduler obeys a **scheduling policy** with two parts:
- **Selection function**
  - Determines which of the ready processes should run on the CPU
  - May be based on priority, resource requirements or the execution characteristics of the process
  - If based on execution characteristics of the process, these can include:
    - time spent waiting, so far;
    - time spent running on the CPU, so far;
    - the process's estimated total service time.
- **Decision mode**
  - Specifies the instants in time at which the selection function is invoked;
  - Two categories:
    - *non-preemptive*: running processes will continue unless they block themselves
    - *preemptive*: running processes may be interrupted and moved to the ready state by the OS

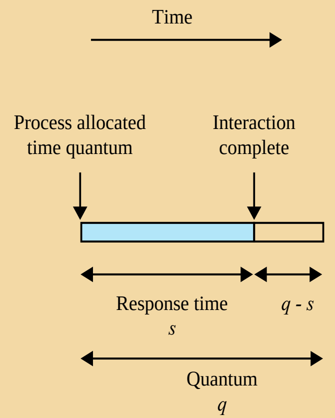
1. First-Come First-Served (FCFS)

- Also known as first-in first-out (FIFO) or a ***strict*** queueing scheme
- Selection function: the steps involved are
  - As each process becomes ready, it joins the ready queue;
  - When the currently running process terminates, the process that has been in the ready queue the longest is selected to run on the CPU;
- Decision mode: non-preemptive, so no (timer) interrupts;
- On its own, **not** ideal for time-sharing systems:
  - in a mix of few long and many short processes, tends to impact the short processes more;
  - tends to benefit CPU bound processes, if there are many more I/O bound processes;
  - tends to impact I/O bound processes, if there are many more CPU bound processes;
  - Can be very inefficient, leaving hardware resources idle;

2. Round Robin

- Decision mode: preemptive, so interrupts allowed
- Preemption based on hardware clock/timer;
- Also known as *time slicing*, because each process is given a "slice" of time with the CPU before being preempted;
- Selection function: upon preempting a process that has used up its time "slice" or the process being blocked/terminating, the next process selected has been waiting the longest in the ready queue;
- A principal design issue is the length of the time quantum (or "slice") to be used;
- Particularly effective policy for time-sharing or transaction processing systems;
- However, tends to impact I/O bound processes that spend a lot of time in "blocked" state waiting for I/O requests to complete;

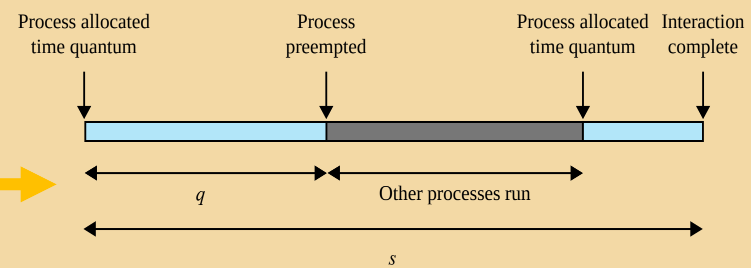
Why Length of "Time Quantum" Matters



(a) Time quantum greater than typical interaction

If length of time quantum is much larger than the total service time s for most processes, then *round robin degenerates to FCFS*

If length of time quantum is much smaller than the total service time s for most processes, then round robin tends to impact average throughput and service time, even though response time is relatively high. As a user, which would you prefer?



(b) Time quantum less than typical interaction

3. Shortest Process Next (SPN)

- Decision mode: non-preemptive policy
- Selection function: the process in the ready queue with the shortest expected processing time is selected next. Use FCFS to break ties;
  - Once selected it runs until it either blocks or terminates;
- A short process will jump to the head of the queue. So, possibility of *starvation* for longer processes;
- A difficulty is the need to know, or at least estimate, the required processing time of each process;
- Programmer’s can give such estimates, but if this turns out to be substantially smaller than the actual running time, the OS may abort the job

4. Shortest Remaining Time (SRT)

- Decision mode: preemptive version of SPN – can preempt running process;
- Selection function: when a process terminates, blocks or joins the ready queue, the short term scheduler chooses the process that has the shortest expected remaining processing time. Use FCFS to break ties;
- Risk of ***starvation*** for longer processes;
- Similar to SPN, SRT requires an estimate of the remaining time required by each process;
- Differs from other policies:
  - Tends to give better throughput than SPN, since shorter jobs never have to wait for longer ones;
  - Does not have the bias FCFS has for long processes;
  - Tends to give better throughput, and better overhead, than round robin;

Two reasons for this:
- no timer interrupts;
- Scheduler only needs to compare newly arrived process with currently running process – other "ready" processes don't need to be compared (why?)

5. Highest Response Ratio Next (HRRN)

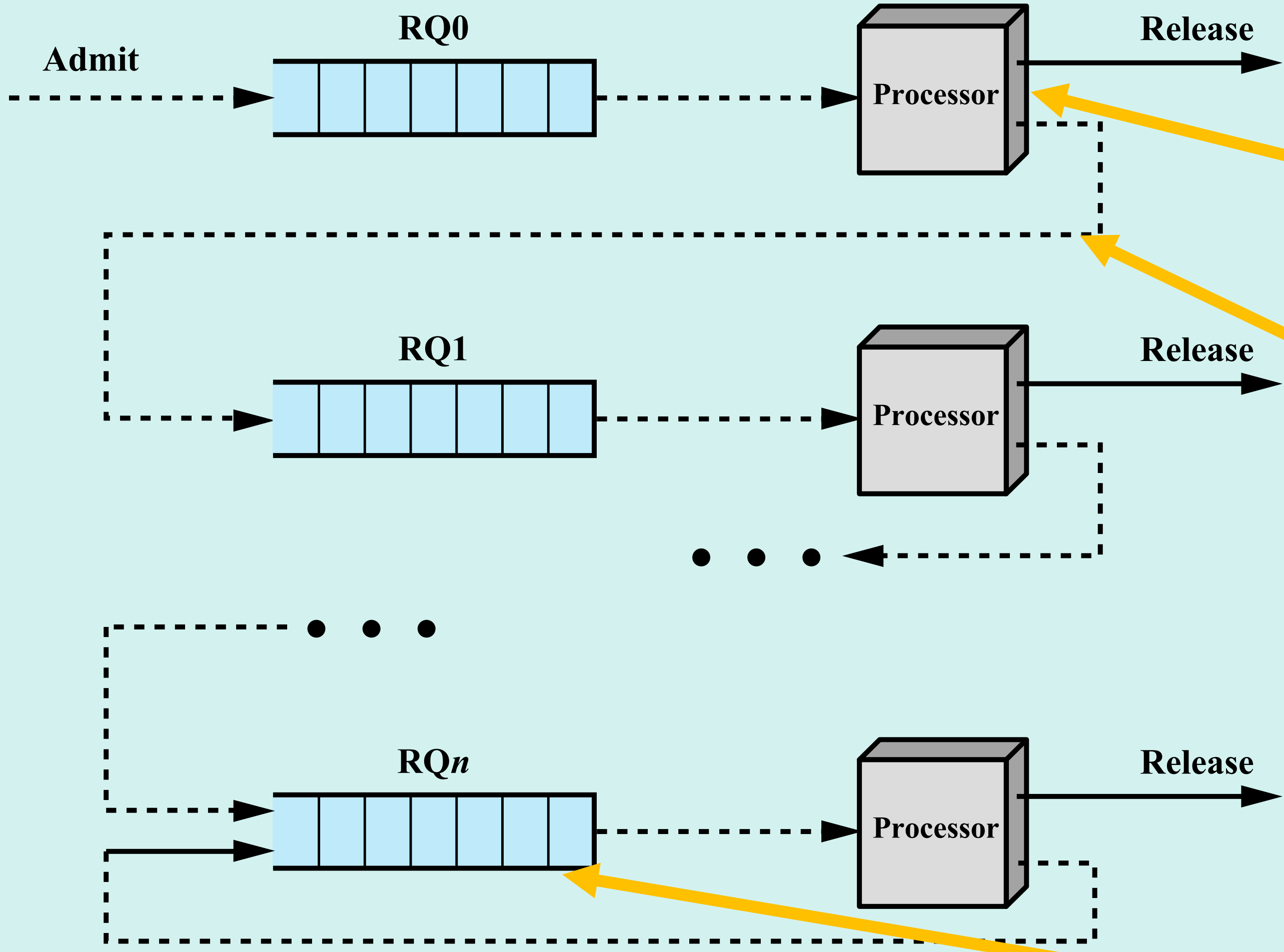
- Selection function: when the current process completes or is blocked, the scheduler chooses the process with the highest ratio (see below). Use FCFS to break ties.
- Decision mode: non-preemptive;
- Attractive because it accounts for the age of the process;
- While shorter jobs are favoured, aging without service increases the ratio so that a longer process will eventually get past competing shorter jobs;

Turnaround time: estimate of the time from when the process first arrived in the ready queue to when it successfully completes.

$$Ratio = \frac{time\ spent\ waiting + expected\ service\ time}{expected\ service\ time}$$

Estimate of time needed by the process (i.e. s), assuming its execution is not preempted.

6. Multilevel Feedback



The decision mode is preemptive: each process assigned to the CPU is given an amount of time to run before it must yield the CPU (like in **round robin**). If the process makes an I/O request, the CPU is relinquished sooner.

When a process is preempted it moves to a lower priority queue. When a process blocks for I/O, it moves to a higher priority queue. Only when all of the processes in a higher priority queue have terminated will lower priority processes be assigned to the CPU (but for longer times).

The longer a CPU-bound process takes to run, the lower the priority of the queue it is assigned to. But, at the same time, these processes are given more time to run when (if?) they are eventually assigned to the CPU. Contrastingly, I/O bound processes migrate to higher priority queues, since these block quite often. And, therefore, they yield the CPU quite often.

Within each queue, the processes are ordered to execute on a **FCFS** basis