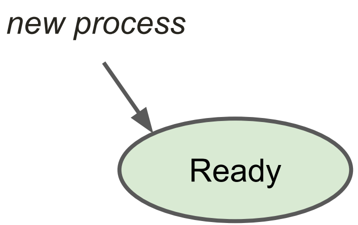
Process Lifecycle
and
Unix Process Creation

UNIVERSITY OF NORTH CAROLINA
CHARLOTTE

Process Lifecycle

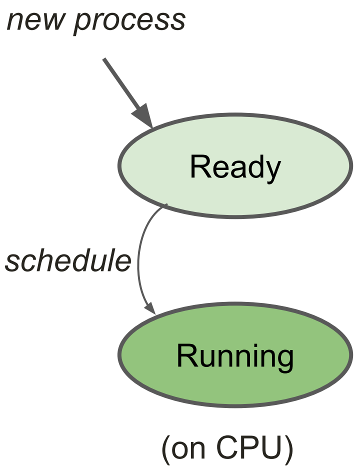
- To execute an instance of a program, *new process* must be created
  - Upon creation, process becomes active or ***ready*** for execution



- Multiple processes can be simultaneously ***ready***
  - E.g., browser, music player, messaging app, etc. can be simultaneously open

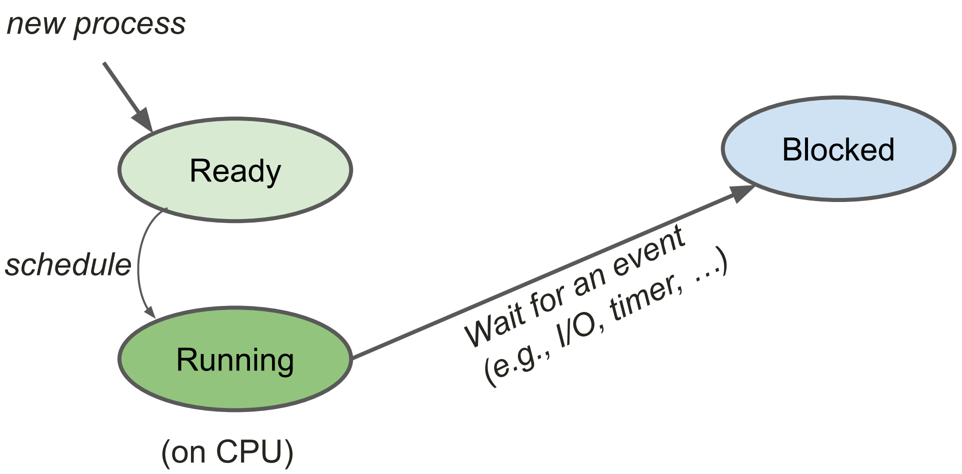
Process Lifecycle

- When CPU is free, OS chooses / schedules a ready process to ***run*** on the CPU
    - The chosen / scheduled process can actively execute instructions on the CPU



Process Lifecycle

- Running process may sometimes need to pause & ***wait*** for some event
  - Process is put into ***blocked*** state so that it doesn't unnecessarily occupy the CPU

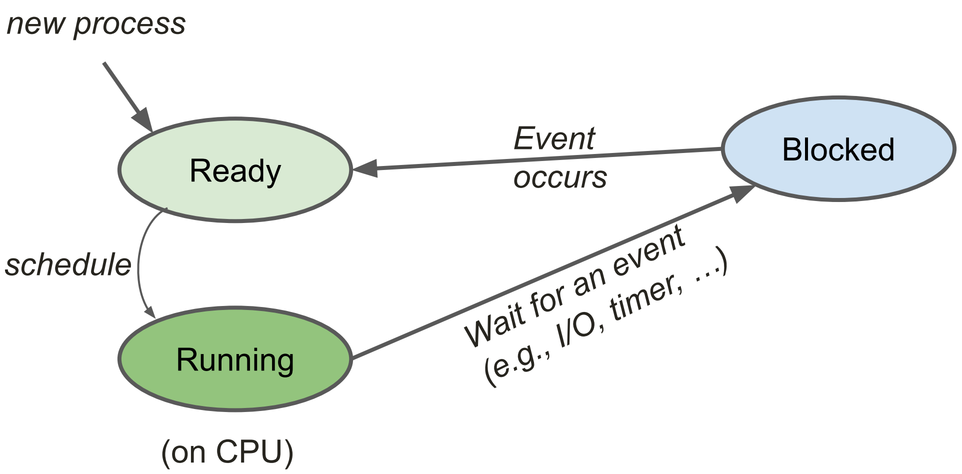


- In the meantime, OS schedules another ***ready*** process to ***run***
  - I.e., OS switches between processes → concept of **multiprogramming**

COLLEGE OF COMPUTING AND INFORMATICS

Process Lifecycle

- Once event that ***blocked*** process is waiting on occurs…
  - …process becomes ***ready*** again

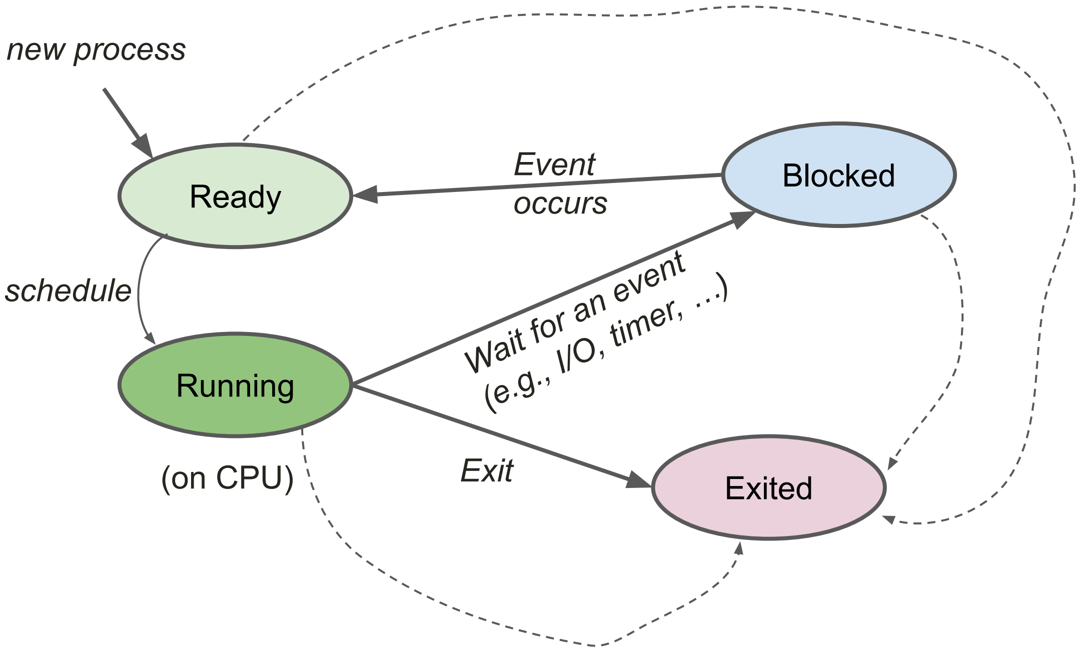


- This *ready – running – blocked* cycle for a process can repeat

COLLEGE OF COMPUTING AND INFORMATICS

Process Lifecycle

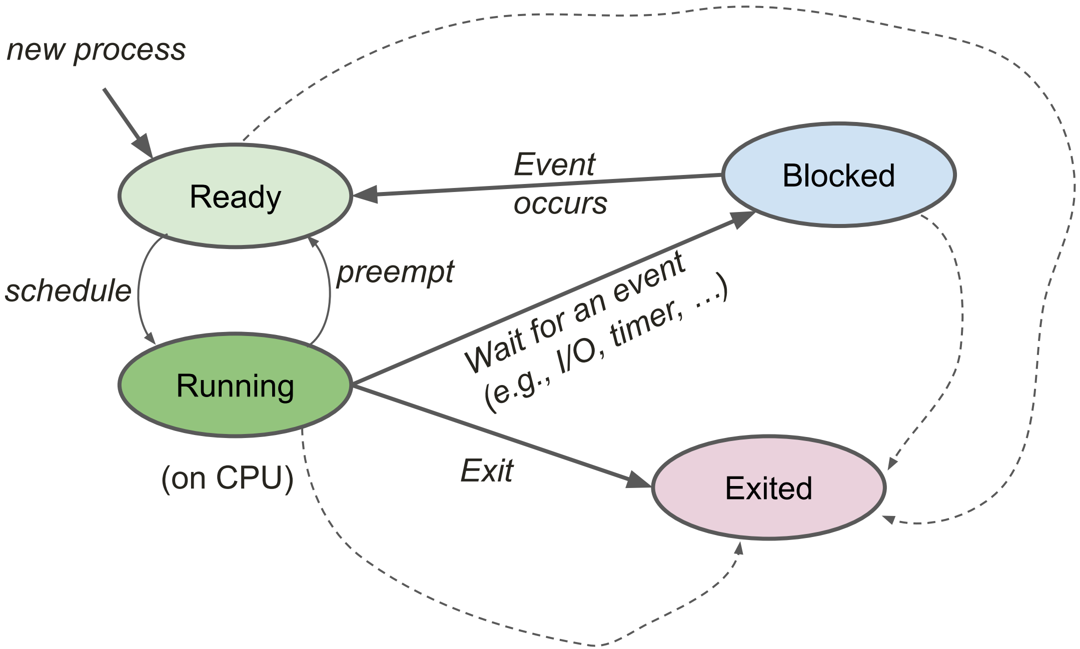
- Once process is done, it ***exits***
- Process can actually exit from any state
  - E.g., user manually kills process, process gets killed due to error, etc.



COLLEGE OF COMPUTING AND INFORMATICS

Process Lifecycle

- While process is *running*, OS may pause/*preempt* it and schedule another process (e.g., due to priority or simply for timesharing)
  - Preempted process goes back to *ready* state



  - Newly scheduled processes moves to *running* state

Process Lifecycle

- Every time execution of running process is paused (due to blocking / preemption)
  - Its execution context needs to be saved so that it can resume later

- Every time a ready process is scheduled
  - Its execution context needs to be loaded / restored for it to run

- → **Context Switching**

To enable multiprogramming & context switching…

- …OS must maintain information & execution context for each process
  - **Process control struct / block** used for this
    - Process ID
    - Process State (ready, running etc.)
    - Program Counter – address of next instruction to be executed
    - Registers – general purpose registers, stack pointer etc.
    - Scheduling information
    - Memory management information
    - Accounting information – time limits, etc.
    - …

Now that we understand the process lifecycle in general, let's discuss process creation and termination in Unix-based systems

Unix process creation (`fork`)

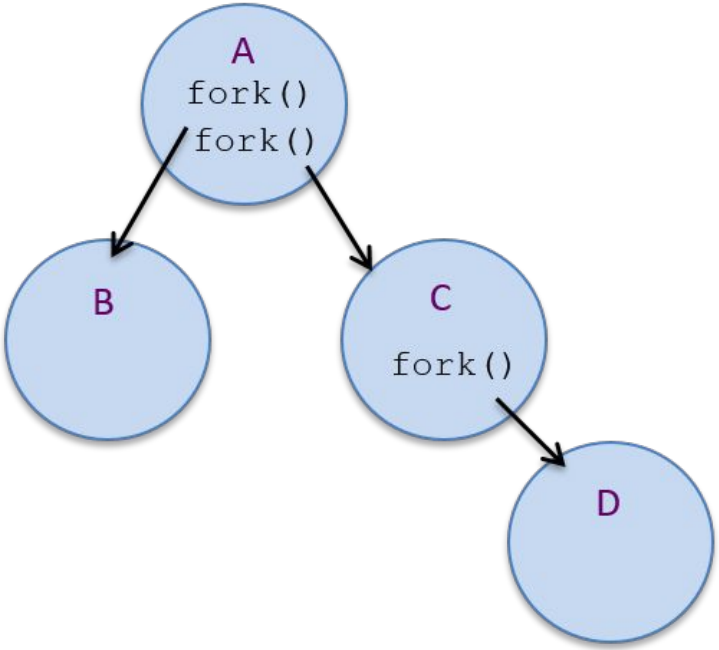
- Existing process makes **`fork`** system call to create *new process*
  - First process (*init*) gets created at system start

- Creating process is the *parent* & new process is the *child*
  - Data from parent process ***copied*** to memory of child process
    - Memory image, environment settings, I/O handles, etc.
  - *Child* gets its own ID, process control block, scheduling info, etc.

- *Hierarchy* of parent-child relationships exist between active processes

Unix process creation (`fork`)

- E.g.: If process A creates two new processes (B and C)...
  - ...and process C in turn creates another new process (D)



Differentiating between parent & child

- If process creation fails, **`fork`** returns -1 to parent & no child is created

- Successful **`fork`** call returns different values to parent and child
  - Fork returns child's process ID to parent process…
  - …and returns 0 to child process

- Parent/child independently continue from execution point immediately after **`fork`**
  - Order of execution of the two processes may vary

```
pid_t id;

id = fork(); /* create new process */
if(id == -1) { /* error creating process */
    printf ("Error creating process\n");
} else if (id == 0) { /* only child process executes this */
    printf ("I'm a child process!\n");
} else { /* only parent process executes this */
    printf ("I just became a parent!\n");
}
```

COLLEGE OF COMPUTING AND INFORMATICS

Switching programs

- Child can *change* the program it executes
  - Invoke **exec** family of system calls to change program that it executes
  - Child *stops* executing code in parent program & *starts* executing new program

```
pid_t id;
char *argv[2];

args[0] = "echo"; /* name of new program for child to execute */
args[1] = "hello"; /* parameter to pass to the above program */
args[2] = NULL; /* end of parameter list */

id = fork(); /* create new process */
if(id == -1) { /* error creating process */
    printf ("Error creating process\n");
} else if (id == 0) { /* only child process executes this */
    execvp(args[0], args); /* child process switches to program indicated by
                              args[0] with parameters in rest of args array */
} else { /* only parent process executes this */
    printf ("I just became a parent!\n");
}
```

COLLEGE OF COMPUTING AND INFORMATICS

Process termination

- As discussed earlier, process may terminate for many reasons
  - Voluntary exit upon task completion
  - Voluntary exit due to fatal error
  - Involuntary exit due to error/bug
  - Involuntary exit due to ***kill*** command by OS or other process who is authorized to do this

On Unix based systems…

- Parent must be allowed to read child's *exit status* or *return value*
  - Parent process can reap children by waiting for them to terminate
    - OS provides system call for this → **`wait`**

- If *parent* terminates before *child…*
  - Child is now an *orphan* process
  - Orphan processes are *adopted* by *init* process

- If a *child* process terminates before *parent*
  - System will still need to keep child's *process control struct*
  - Child process becomes a *zombie* process
    - "Dead", but not "reaped"

Waiting for child to terminate

```
pid_t id, ret;
int status;

id = fork(); /* create new process */
if(id == -1) { /* error creating process */
    printf ("Error creating process\n");
} else if (id == 0) { /* only child process executes this */
    char * args[] = {"echo", "hello", NULL};
    execvp(args[0], args); /* child process switches to program indicated by
                            args[0] with parameters in rest of args array */
} else { /* only parent process executes this */
    printf ("I just became a parent!\n");
    ret = wait(&status); /* parent process waits for child to terminate */
}
```

COLLEGE OF COMPUTING AND INFORMATICS