ILLINOIS TECH

College of Computing

CS 450 Operating Systems

Process (cont.)

Yue Duan

Process

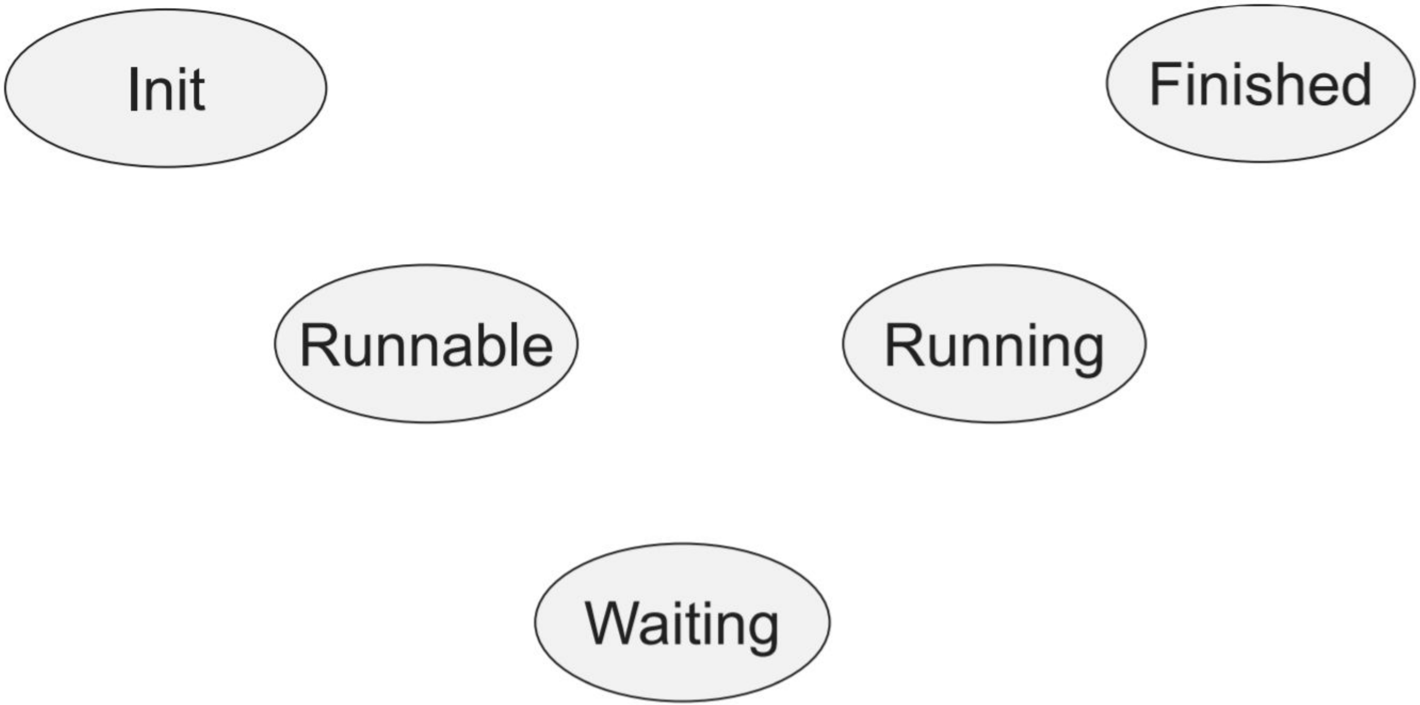
- A process physically runs on the CPU
  - dozens of processes running at the same time
    - e.g., browser, music player, chatting app, system services
  - each process has its own
    - Registers
    - Memory
    - I/O resources
    - “thread of control”
- need to multiplex, schedule, … to create virtual CPUs for each process

For now, assume we have a single core CPU

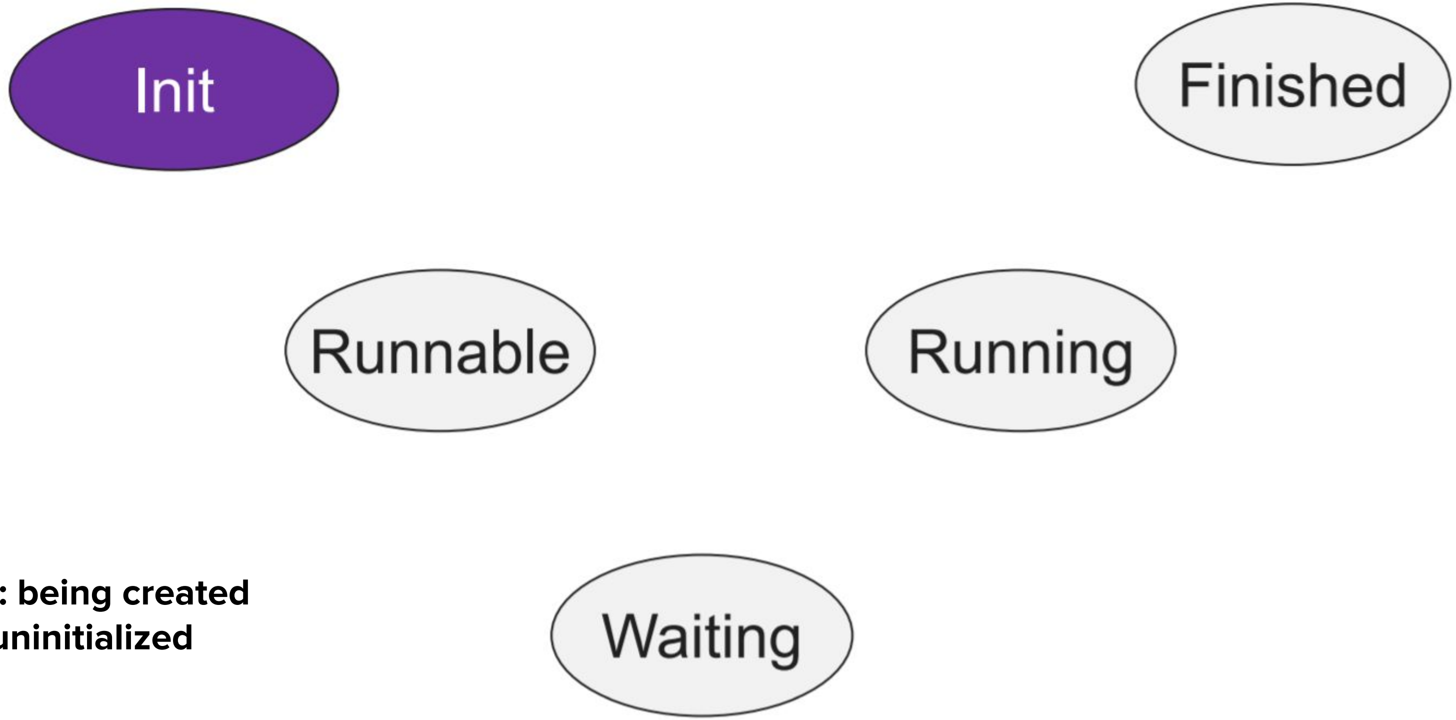
Process Control Block (PCB)

- For each process, OS has a **Process Control Block** (PCB):
  - location in memory (page table)
  - location of executable on disk
  - which user is executing this process (uid)
  - process identifier (pid)
  - process status (running, waiting, finished, etc.)
  - scheduling information
  - interrupt stack
  - saved kernel SP (when process is not running)
    - points into interrupt stack
    - interrupt stack
  - ... and more!

Process Life Cycle

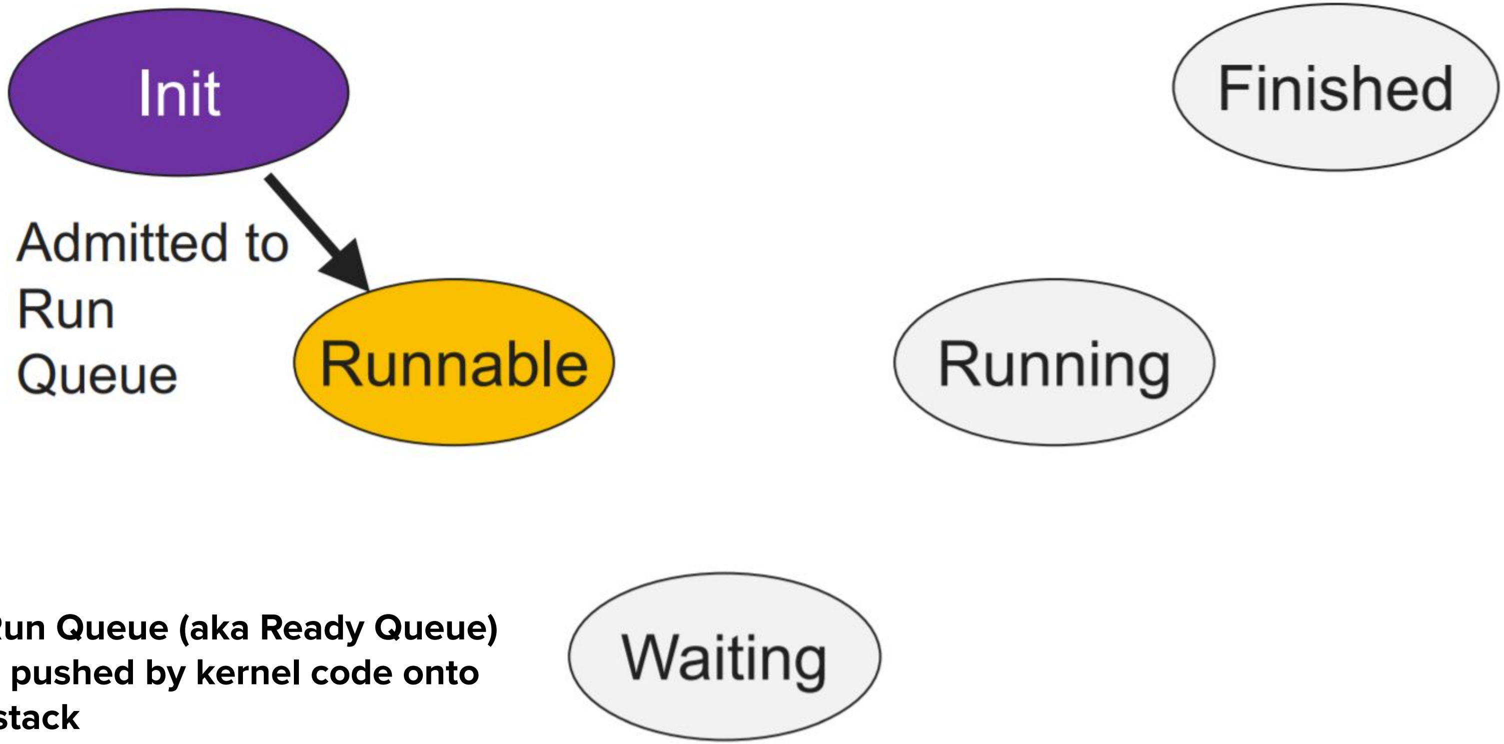


Process Life Cycle: Process creation



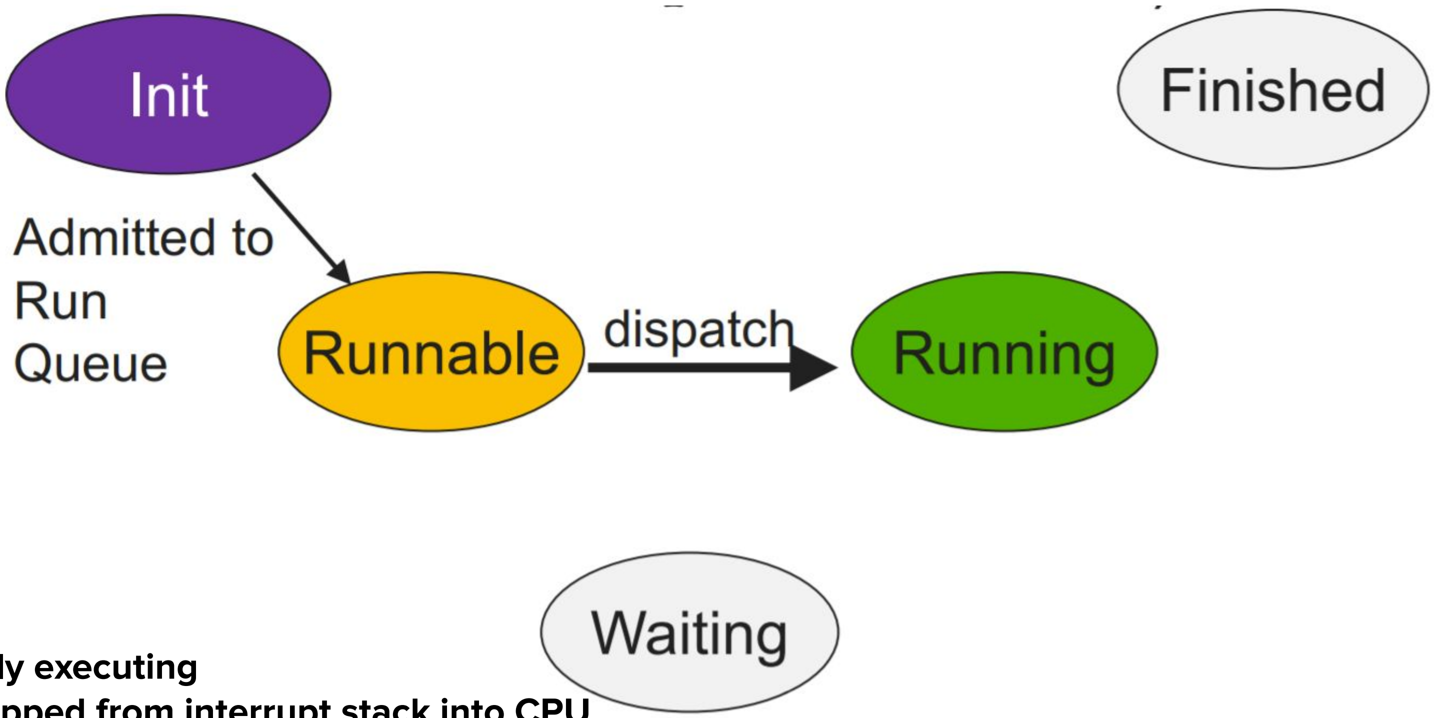
PCB status: being created
Registers: uninitialized

Process Life Cycle: Process is Ready to Run



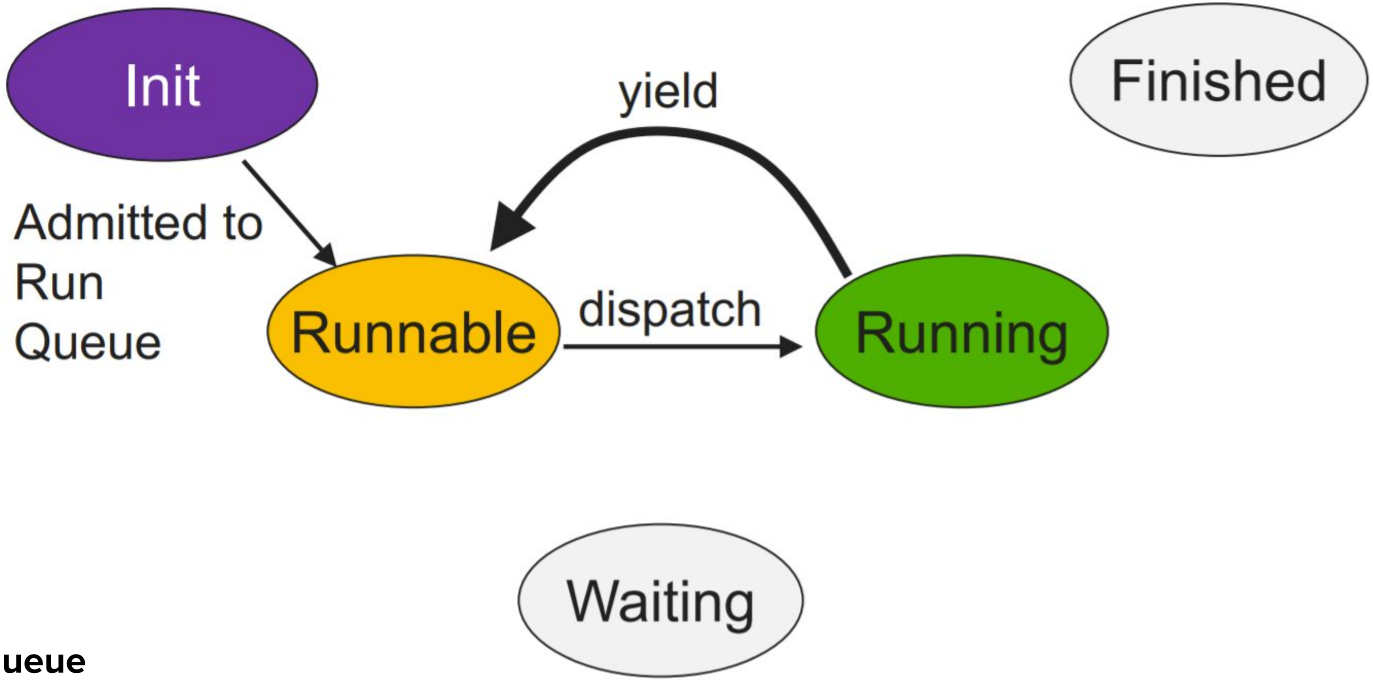
PCB: on Run Queue (aka Ready Queue)
Registers: pushed by kernel code onto interrupt stack

Process Life Cycle: Process is Running



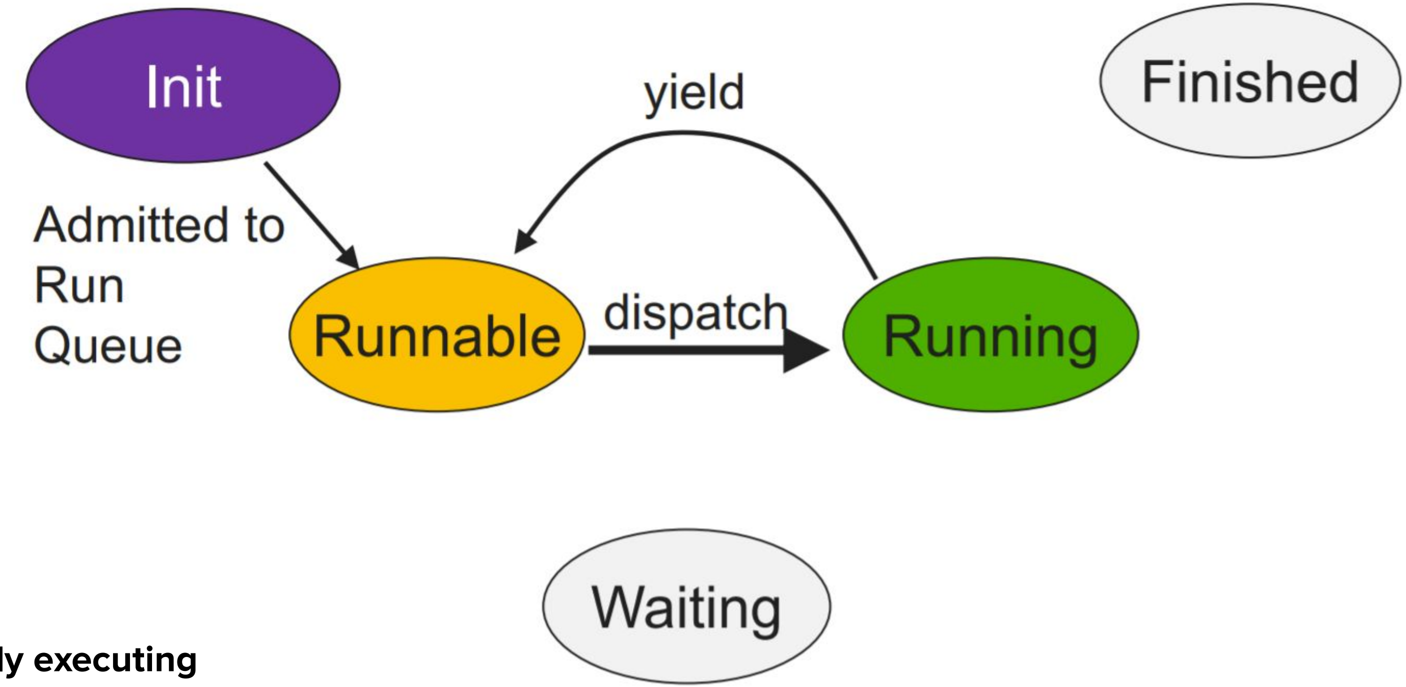
PCB: currently executing
Registers: popped from interrupt stack into CPU

Process Life Cycle: Process Yields



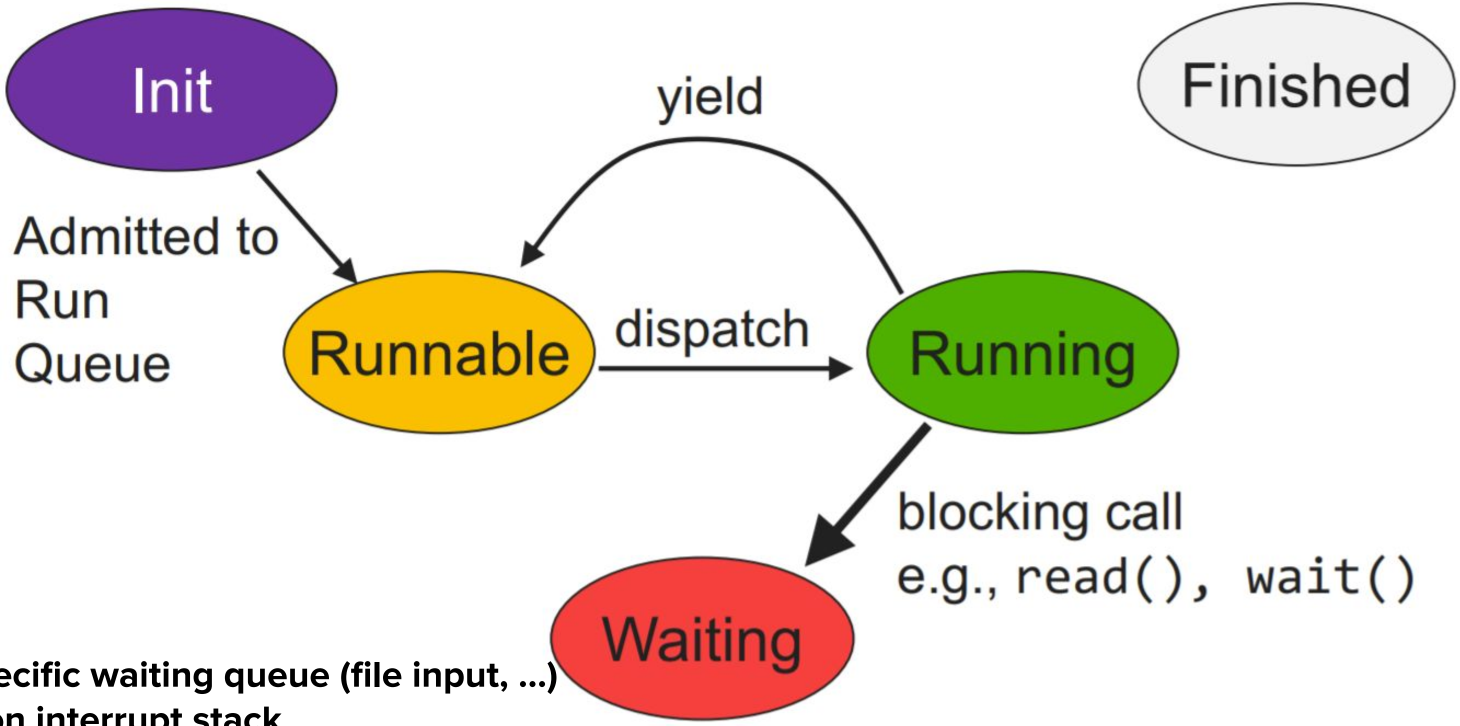
PCB: on Run queue
Registers: pushed onto interrupt stack (sp saved in PCB)

Process Life Cycle: Process is Running Again



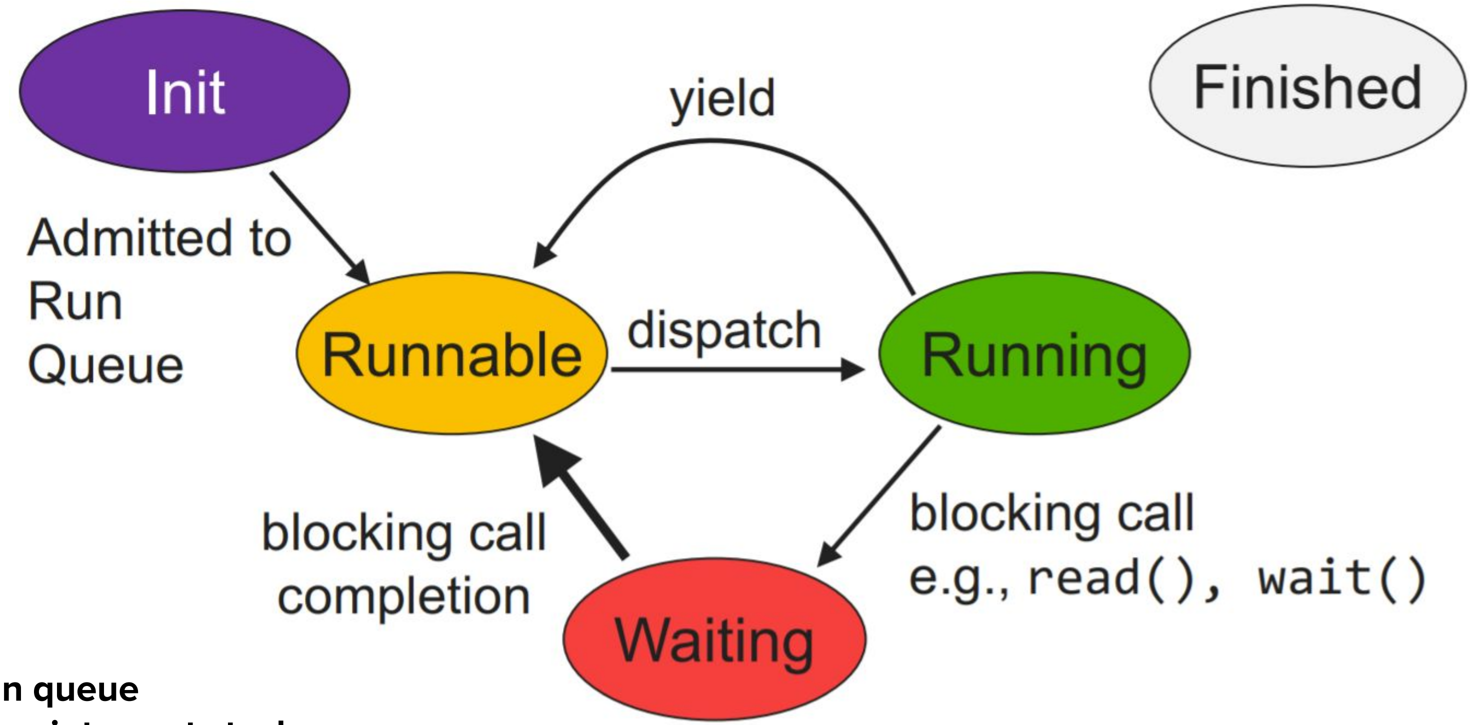
PCB: currently executing
Registers: sp restored from PCB; others restored from stack

Process Life Cycle: Process is Waiting



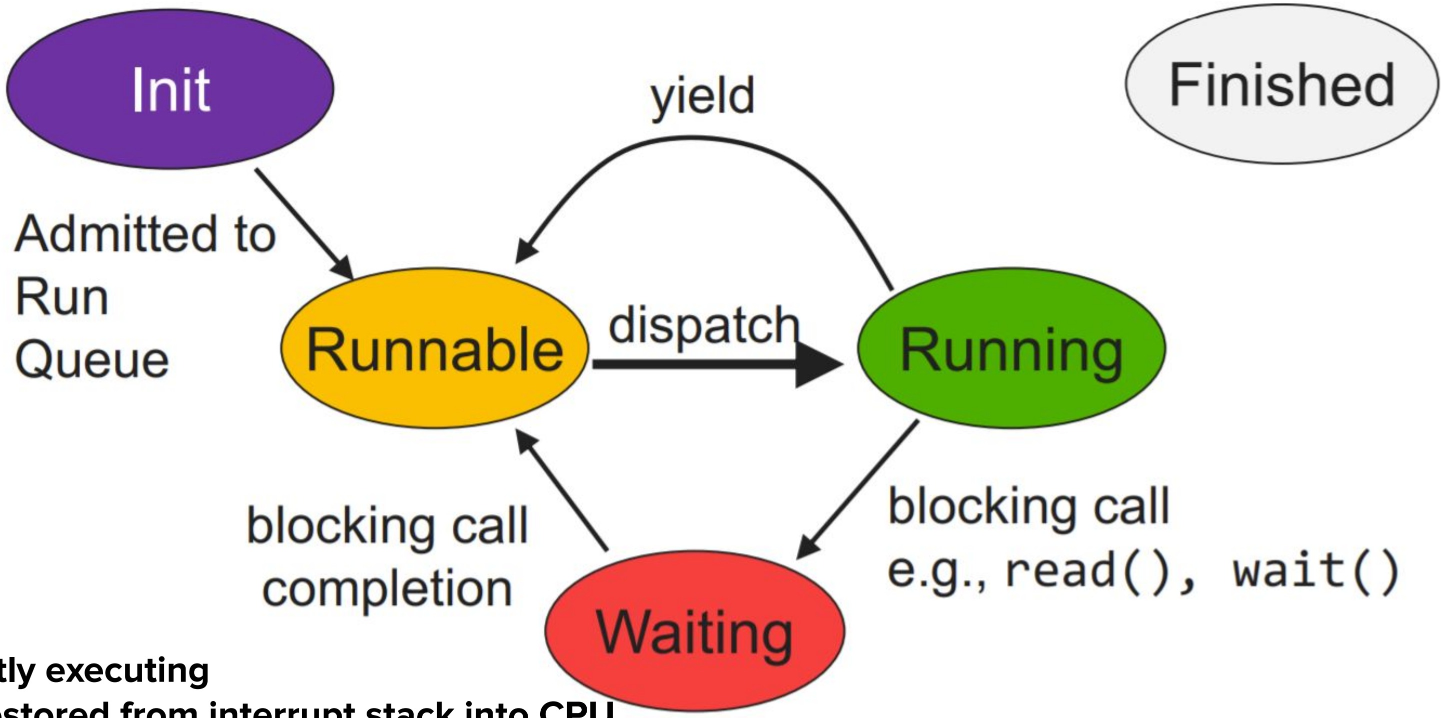
PCB: on specific waiting queue (file input, ...)
Registers: on interrupt stack

Process Life Cycle: Process is Ready Again!



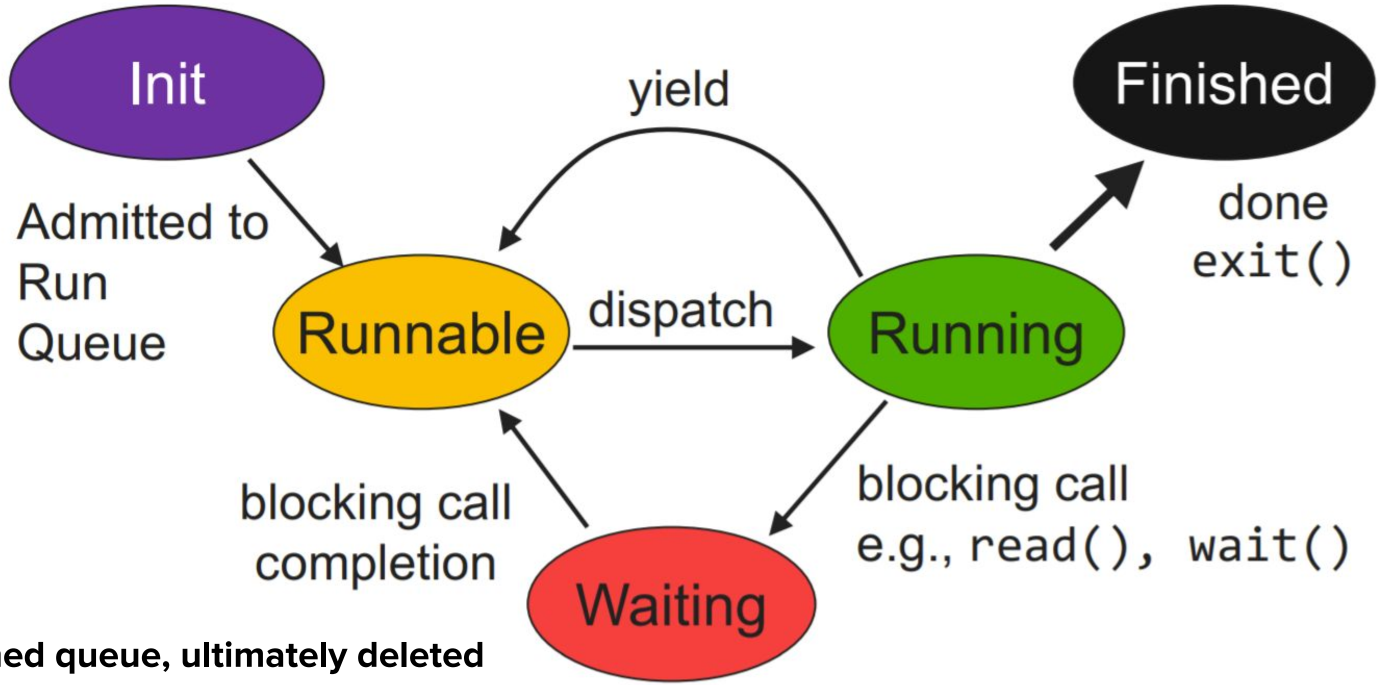
PCB: on run queue
Registers: on interrupt stack

Process Life Cycle: Process is Running Again!



PCB: currently executing
Registers: restored from interrupt stack into CPU

Process Life Cycle: Process is Finished (Process = Zombie)



PCB: on Finished queue, ultimately deleted
Registers: no longer needed

Invariants to keep in mind

- At most 1 process is **RUNNING** at any time (per core)
- When CPU is in user mode, current process is **RUNNING** and its interrupt stack is empty
- If process is **RUNNING**
  - its PCB is not on any queue
- If process is **RUNNABLE** or **WAITING**
  - its interrupt stack is non-empty and can be switched to
    - i.e., has its registers saved on top of the stack
  - its PCB is either
    - on the run queue (if **RUNNABLE**)
    - on some wait queue (if **WAITING**)
- If process is FINISHED - its PCB is on finished queue

Cleaning up zombies

- Process cannot clean up itself
  - WHY NOT?
- Process can be cleaned up
  - either by any other process
    - check for zombies just before returning to RUNNING state
  - or by parent when it waits for it - but what if the parent dies first?
  - or by dedicated “reaper” process
- Linux uses combination:
  - usually parent cleans up child process when waiting
  - if parent dies before child, child process is inherited by the initial process, which is continuously waiting

Switching to another process

- Switching from executing the current process to another runnable process
  - Process 1 goes from RUNNING ===> RUNNABLE/WAITING
  - Process 2 goes from RUNNABLE ===> RUNNING
- 1. save kernel registers of process 1 on its interrupt stack
- 2. save kernel sp of process 1 in its PCB
- 3. restore kernel sp of process 2 from its PCB
- 4. restore kernel registers from its interrupt stack

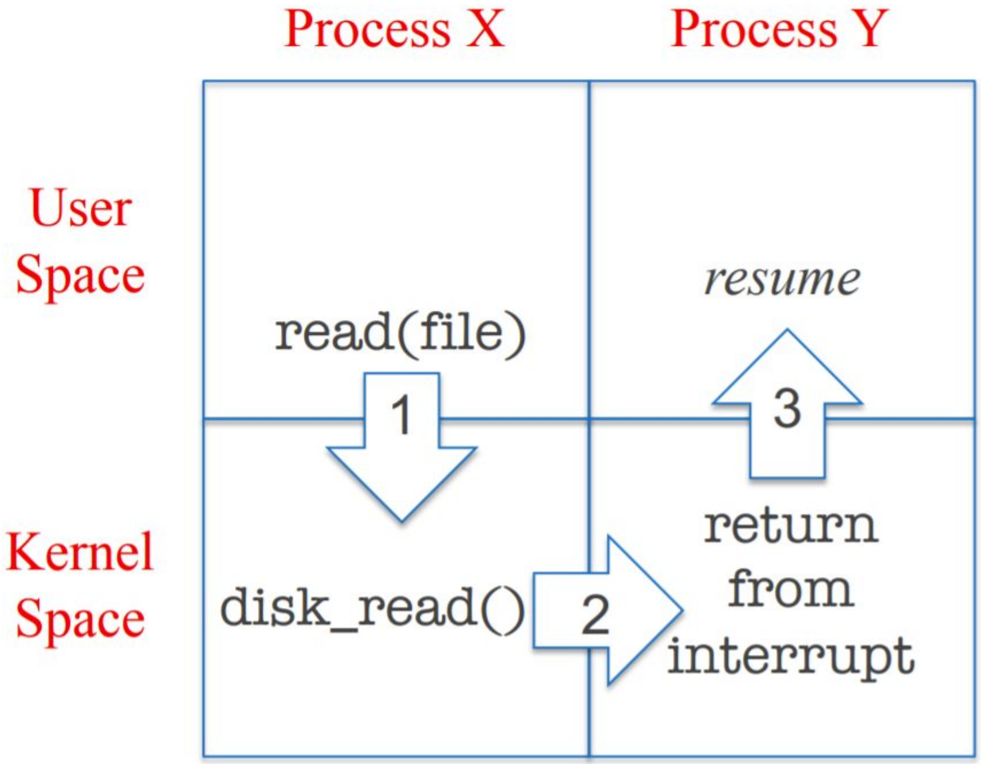
Switching to another process

- What if there are no more RUNNABLE processes?
    - scheduler() would return NULL and things blow up
    - solution: always run a low priority process that sits in an infinite loop executing the x86 HLT instruction
        - which waits for the next interrupt, saving energy when there's nothing to do
    - Interrupt handler should yield() if some other process is put on the run queue

Context switches

- 1. Interrupt: From user to kernel space
  - system call, exception, or interrupt
- 2. Yield: between two processes
  - happens inside the kernel, switching from one PCB/interrupt stack to another
- 3. From kernel space to user space
  - through a return_from_interrupt
- Note that each involves a stack switch:
  - 1. Px user stack ===> Px interrupt stack
  - 2. Px interrupt stack ===> Py interrupt stack
  - 3. Py interrupt stack ===> Py user stack

Example switch between processes



- 1. save process X user registers
- 2. save process X kernel registers and restore process Y kernel registers
- 3. restore process Y user registers

System calls to create a new process

- Windows CreateProcess (simplified)

```
if (!CreateProcess(
    NULL,  // No module name (use command line)
    argv[1],// Command line
    NULL,  // Process handle not inheritable
    NULL,  // Thread handle not inheritable
    FALSE, // Set handle inheritance to FALSE
    0,     // No creation flags
    NULL,  // Use parent's environment block
    NULL,  // Use parent's starting directory
    &si,   // Pointer to STARTUPINFO structure
    &pi )  // Ptr to PROCESS_INFORMATION structure
)
```

System calls to create a new process

- Linux fork (actual form)

```
int pid = fork(  void ☺
    NULL,   // No module name (use command line)
    argv[1],// Command line
    NULL,   // Process handle not inheritable
    NULL,   // Thread handle not inheritable
    FALSE,  // Set handle inheritance to FALSE
    0,      // No creation flags
    NULL,   // Use parent's environment block
    NULL,   // Use parent's starting directory
    &si,    // Pointer to STARTUPINFO structure
    &pi )
)
```

pid = process identifier

Kernel actions to create a process

- fork():
    - Allocate ProcessID
    - Create & initialize PCB
    - Create and initialize a new address space
    - Inform scheduler that new process is ready to run

- exec (program, arguments):
    - Load the program into the address space
    - Copy arguments into memory in address space
    - Initialize h/w context to start execution at “start”

Creating and Managing Processes

`fork()`	Create a child process as a clone of the current process. Returns to both parent and child. Returns child pid to parent process, 0 to child process.
`exec (prog, args)`	Run the application **prog** in the current process with the specified arguments (*replacing any code and data that was in the process already*)
`wait (&status)`	Pause until a child process has exited
`exit (status)`	Tell the kernel the current process is complete and should be garbage collected.
`kill (pid, type)`	Send an interrupt of a specified type to a process. (a bit of a misnomer, no?)

Fork + Exec

Process 1
Program A

PC →

```
child_pid = fork();
if (child_pid==0)
  exec(B);
else
  wait(&status);
```

child_pid ?

Fork + Exec

Process 1
Program A

```
child_pid = fork();
if (child_pid==0)
  exec(B);
else
  wait(&status);
```

PC → if (child_pid==0)

child_pid 42

Process 42
Program A

```
child_pid = fork();
if (child_pid==0)
  exec(B);
else
  wait(&status);
```

PC → if (child_pid==0)

child_pid 0

fork returns twice!

Fork + Exec

Process 1
Program A

```
   child_pid = fork();
   if (child_pid==0)
     exec(B);
   else
PC →  wait(&status);
```

child_pid 42

Waits until child exits

Process 42
Program A

```
   child_pid = fork();
PC → if (child_pid==0)
     exec(B);
   else
     wait(&status);
```

child_pid 0

Fork + Exec

Process 1
Program A

```
   child_pid = fork();
   if (child_pid==0)
     exec(B);
   else
PC   wait(&status);
```

child_pid 42

Process 42
Program A

```
   child_pid = fork();
   if (child_pid==0)
PC   exec(B);
   else
     wait(&status);
```

child_pid 0

if and else both executed!

Fork + Exec

Process 1
Program A

```
child_pid = fork();
if (child_pid==0)
  exec(B);
else
  wait(&status);
```

PC

child_pid 42

Process 42
Program B

```
main() {
   ...

   exit(3);
}
```

PC

Fork + Exec

Process 1
Program A

```
child_pid = fork();
if (child_pid==0)
  exec(B);
else
  wait(&status);
```

PC → wait(&status);

child_pid: 42

status: 3

Signals

Allow applications to behave like operating systems.

ID	Name	Default Action	Corresponding Event
2	SIGINT	Terminate	Interrupt (e.g., ctrl-c from keyboard)
9	SIGKILL	Terminate	Kill program (cannot override or ignore)
14	SIGALRM	Terminate	Timer signal
17	SIGCHLD	Ignore	Child stopped or terminated
20	SIGTSTP	Stop until next SIGCONT	Stop signal from terminal (e.g. ctrl-z from keyboard)

Sending a Signal

- Kernel delivers a signal to a destination process
- For one of the following reasons:
  - Kernel detected a system event (e.g., div-by-zero (SIGFPE) or termination of a child (SIGCHLD))
  - A process invoked the kill system call requesting kernel to send signal to a process
    - debugging
    - suspension
    - resumption
    - timer expiration

Receiving a Signal

- A destination process receives a signal when it is forced by the kernel to react in some way to the delivery of the signal.
- Three possible ways to react:
  - 1. Ignore the signal (do nothing)
  - 2. Terminate process (+ optional core dump)
  - 3. Catch the signal by executing a user-level function called signal handler
    - Like a hardware exception handler being called in response to an asynchronous interrupt

Signal Example

```c
int main() {
   pid_t pid[N];
   int i, child_status;

   for (i = 0; i < N; i++) // N forks
       if ((pid[i] = fork()) == 0) {
          while(1); //child infinite loop
       }
   /* Parent terminates the child processes */
   for (i = 0; i < N; i++) {     // parent continues executing
       printf("Killing proc. %d\n", pid[i]);
       kill(pid[i], SIGINT);
   }
   /* Parent reaps terminated children */
   for (i = 0; i < N; i++) {
       pid_t wpid = wait(&child_status);
       if (WIFEXITED(child_status)) // parent checks for each child’s exit
           printf("Child %d terminated w/exit status %d\n", wpid,
                   WEXITSTATUS(child_status));
       else
           printf("Child %d terminated abnormally\n", wpid);
   }
   exit(0);
}
```

Handler Example

```c
void int_handler(int sig) {
   printf("Process %d received signal %d\n", getpid(), sig);
   exit(0);
}
int main() {
   pid_t pid[N];
   int i, child_status;
   signal(SIGINT, int_handler); //register handler for SIGINT
   for (i = 0; i < N; i++) // N forks
       if ((pid[i] = fork()) == 0) {
          while(1); //child infinite loop
       }
   for (i = 0; i < N; i++) {    // parent continues executing
       printf("Killing proc. %d\n", pid[i]);
       kill(pid[i], SIGINT);
   }
   for (i = 0; i < N; i++) {
       pid_t wpid = wait(&child_status);
       if (WIFEXITED(child_status)) // parent checks for each child's exit
           printf("Child %d terminated w/exit status %d\n", wpid,
                   WEXITSTATUS(child_status));
       else
           printf("Child %d terminated abnormally\n", wpid);
   }
   exit(0);
}
```

THANK YOU!