Processes

CS31, Week 10

Prof. Jason Waterman

November 15, 2012

Topics for today

- The UNIX process abstraction
- Process lifecycle
  - Creating processes: forking
  - Running new programs within a process
  - Terminating and reaping processes
- Signaling processes

What is a process?

What is a process?

- A **process** is the OS's abstraction for **execution**
  - A process is *an instance of a program in execution*.
  - i.e., each process is running a program; there may be many processes running the same program

What is a process?

- A **process** is the OS's abstraction for **execution**
  - A process is *an instance of a program in execution*.
  - i.e., each process is running a program; there may be many processes running the same program
- A process provides
  - Private address space
    - Through the mechanism of virtual memory!
  - Illusion of exclusive use of processor

Process context

Process context

- Process **context** is the state that the operating system needs to run a process

Process context

- Process **context** is the state that the operating system needs to run a process
  - 1) **Address space**
    - The memory that the process can access
    - Consists of various pieces: the program code, global/static variables, heap, stack, etc.

Process context

- Process **context** is the state that the operating system needs to run a process
  - 1) **Address space**
    - The memory that the process can access
    - Consists of various pieces: the program code, global/static variables, heap, stack, etc.
  - 2) **Processor state**
    - The CPU registers associated with the running process
    - Includes general purpose registers, program counter, stack pointer, etc.

Process context

- Process **context** is the state that the operating system needs to run a process
  - 1) **Address space**
    - The memory that the process can access
    - Consists of various pieces: the program code, global/static variables, heap, stack, etc.
  - 2) **Processor state**
    - The CPU registers associated with the running process
    - Includes general purpose registers, program counter, stack pointer, etc.
  - 3) **OS resources**
    - Various OS state associated with the process
    - Examples: page table, file table, network sockets, etc.

Context switches

Context switches

- Multiple processes can run simultaneously.
  - On a single CPU system, only one process is running on the CPU at a time.
    - But can have **concurrent** execution of processes
  - On a multi-CPU (or multi-core) system, multiple processes can run in **parallel**.
  - The OS will timeshare each CPU/core, rapidly switching processes across them all.

Context switches

- Multiple processes can run simultaneously.
  - On a single CPU system, only one process is running on the CPU at a time.
    - But can have **concurrent** execution of processes
  - On a multi-CPU (or multi-core) system, multiple processes can run in **parallel**.
  - The OS will timeshare each CPU/core, rapidly switching processes across them all.
- Switching a CPU from running one process to another is called a **context switch**.
  - (1) Save the context of the currently running process,
  - (2) Restore the context of some previously preempted process
  - (3) Resume execution of the newly restored process

Context switches

- Multiple processes can run simultaneously.
  - On a single CPU system, only one process is running on the CPU at a time.
    - But can have **concurrent** execution of processes
  - On a multi-CPU (or multi-core) system, multiple processes can run in **parallel**.
  - The OS will timeshare each CPU/core, rapidly switching processes across them all.
- Switching a CPU from running one process to another is called a **context switch**.
  - (1) Save the context of the currently running process,
  - (2) Restore the context of some previously preempted process
  - (3) Resume execution of the newly restored process
- Deciding when to preempt current process and restart previously preempted process is known as **scheduling**
  - Performed by part of the OS called a **scheduler**

Process IDs

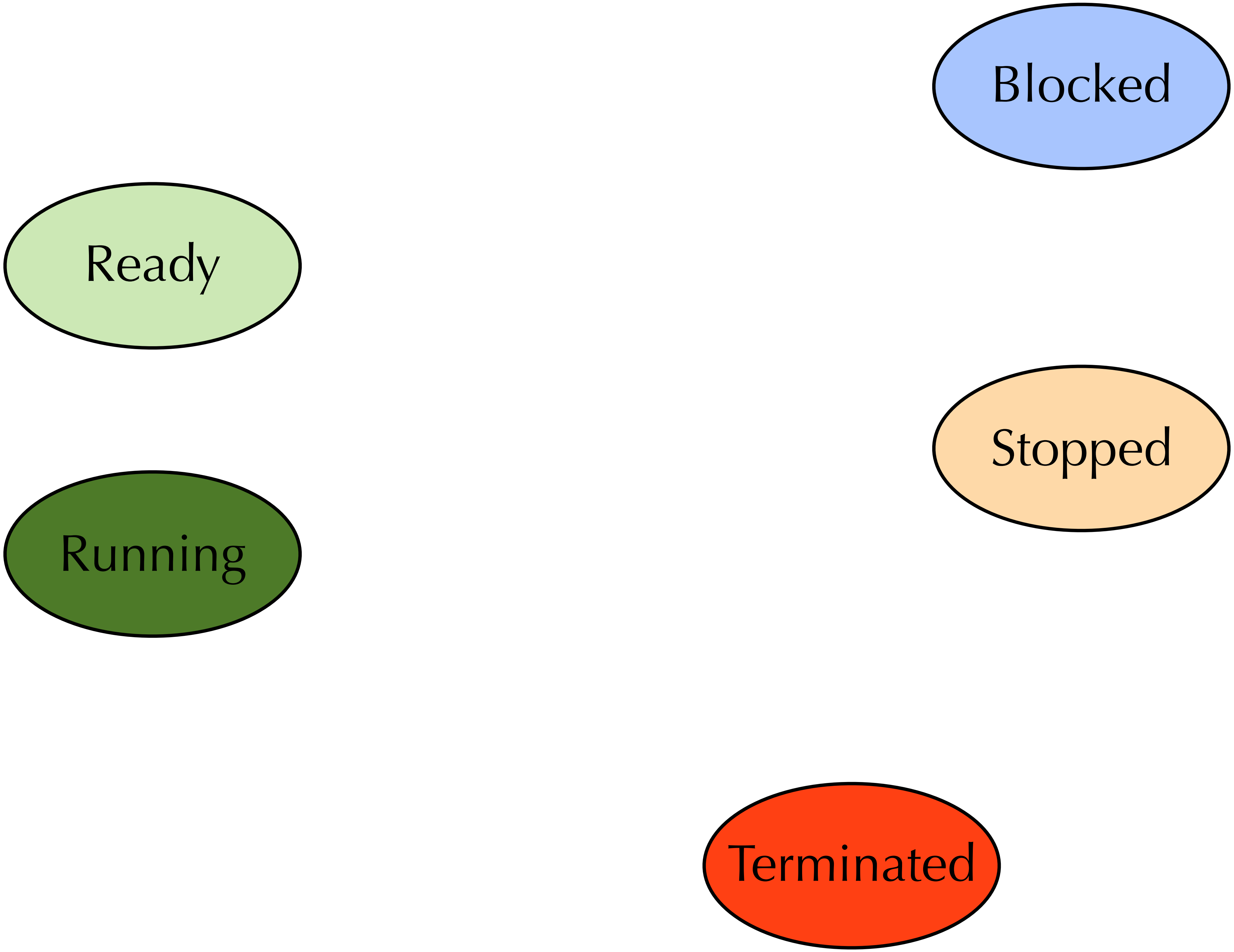
- Each process has a unique positive **process ID (PID)**
- `getpid` returns current process's PID
- `getppid` returns PID of parent of current process

```
pid_t getpid(void);
pid_t getppid(void);
```

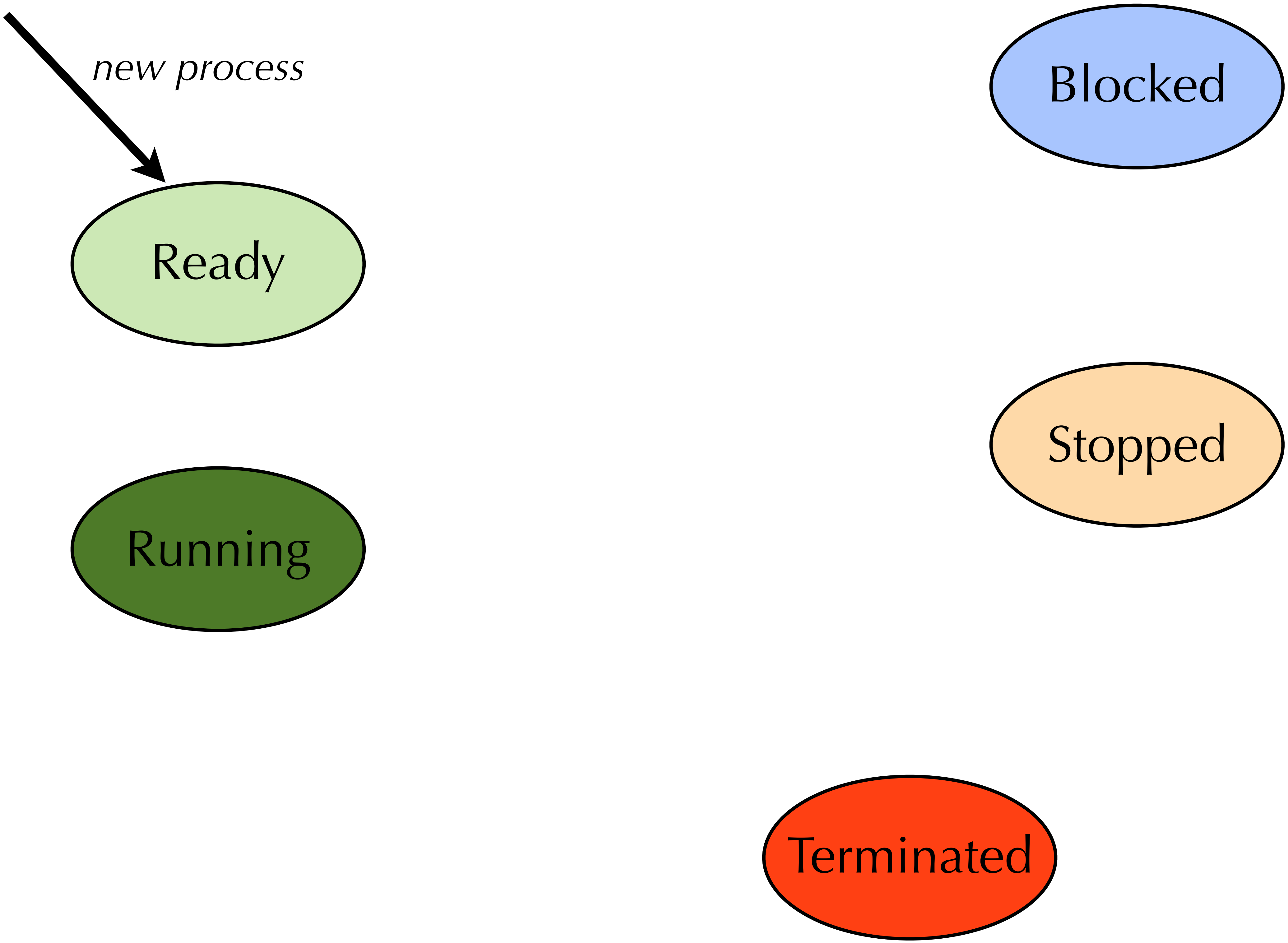
Process states

- At any moment, a process is in one of several states:
  - **Ready**:
    - Process is waiting to be executed
  - **Running**:
    - Process is executing on a CPU
  - **Stopped**:
    - Process is *suspended* (due to receiving a certain **signal**) and will not be scheduled
    - More on signals soon…
  - **Waiting** (or **sleeping** or **blocked**):
    - Process is waiting for an event to occur, such as completion of I/O, timer, etc.
    - Why is this different than “ready” ?
  - **Terminated:**
    - Process is stopped permanently, e.g., by returning from `main`, or by calling `exit`
- As the process executes, it moves between these states
  - What state is the process in most of the time?

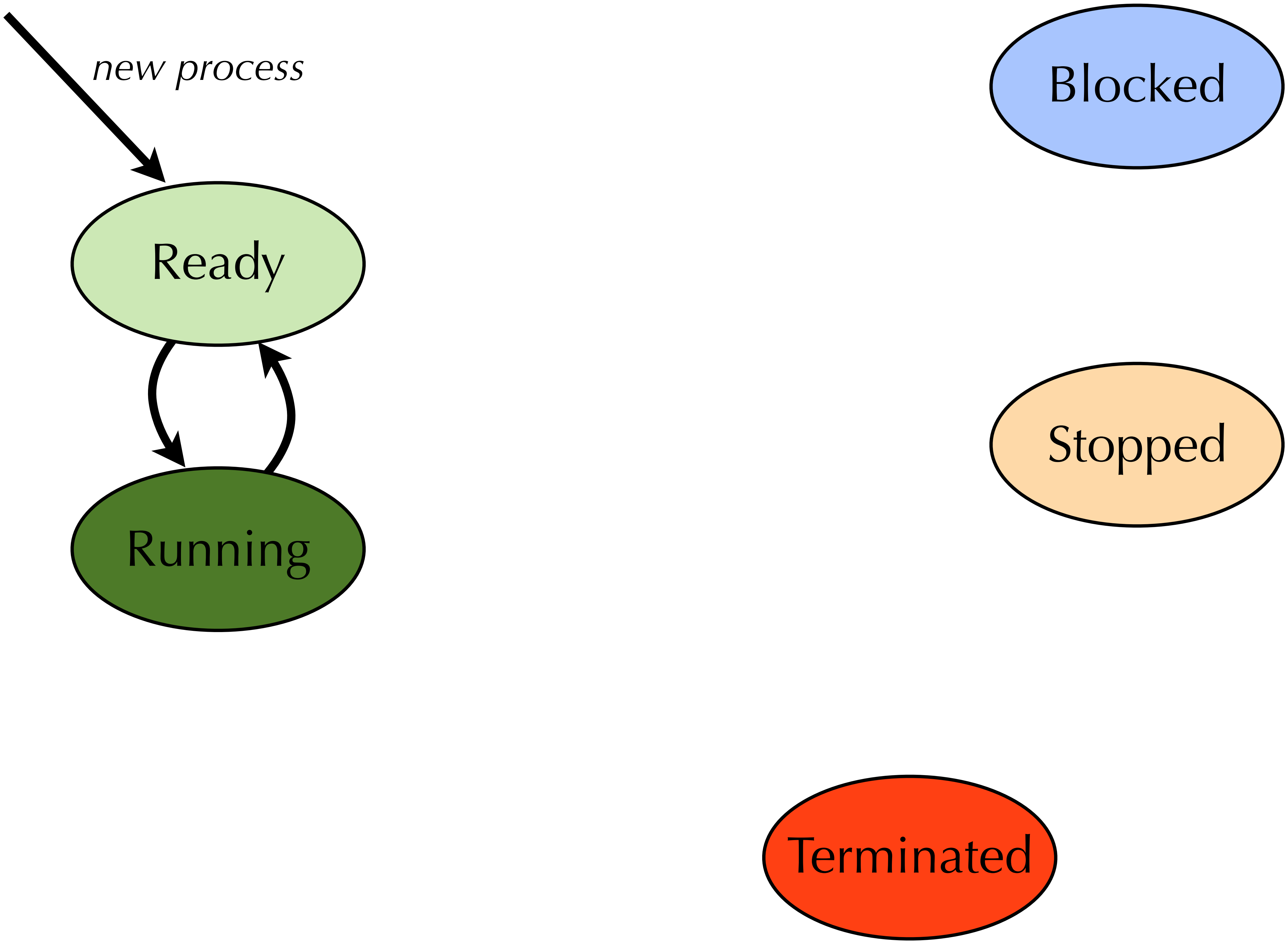
Process lifecycle



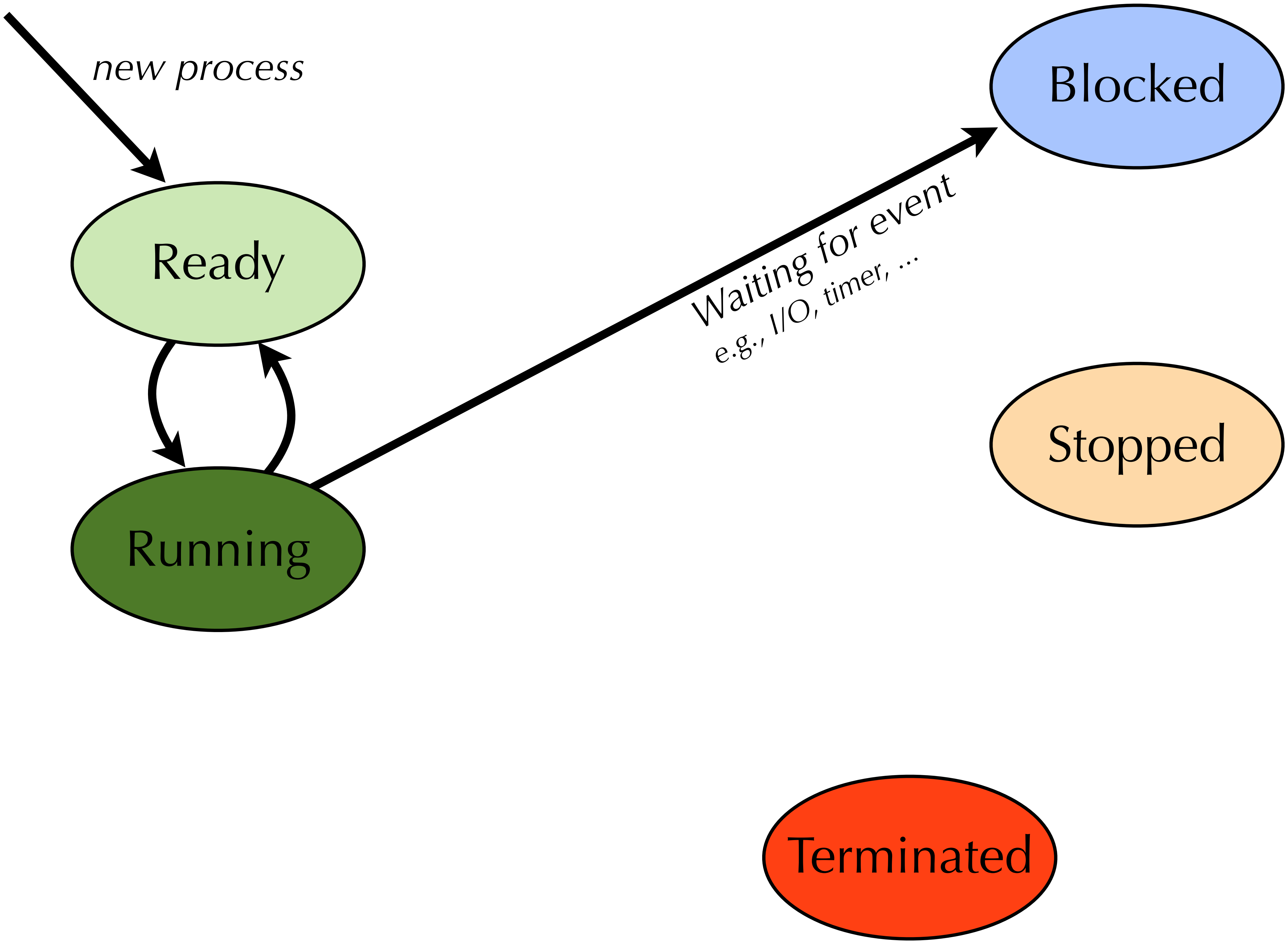
Process lifecycle



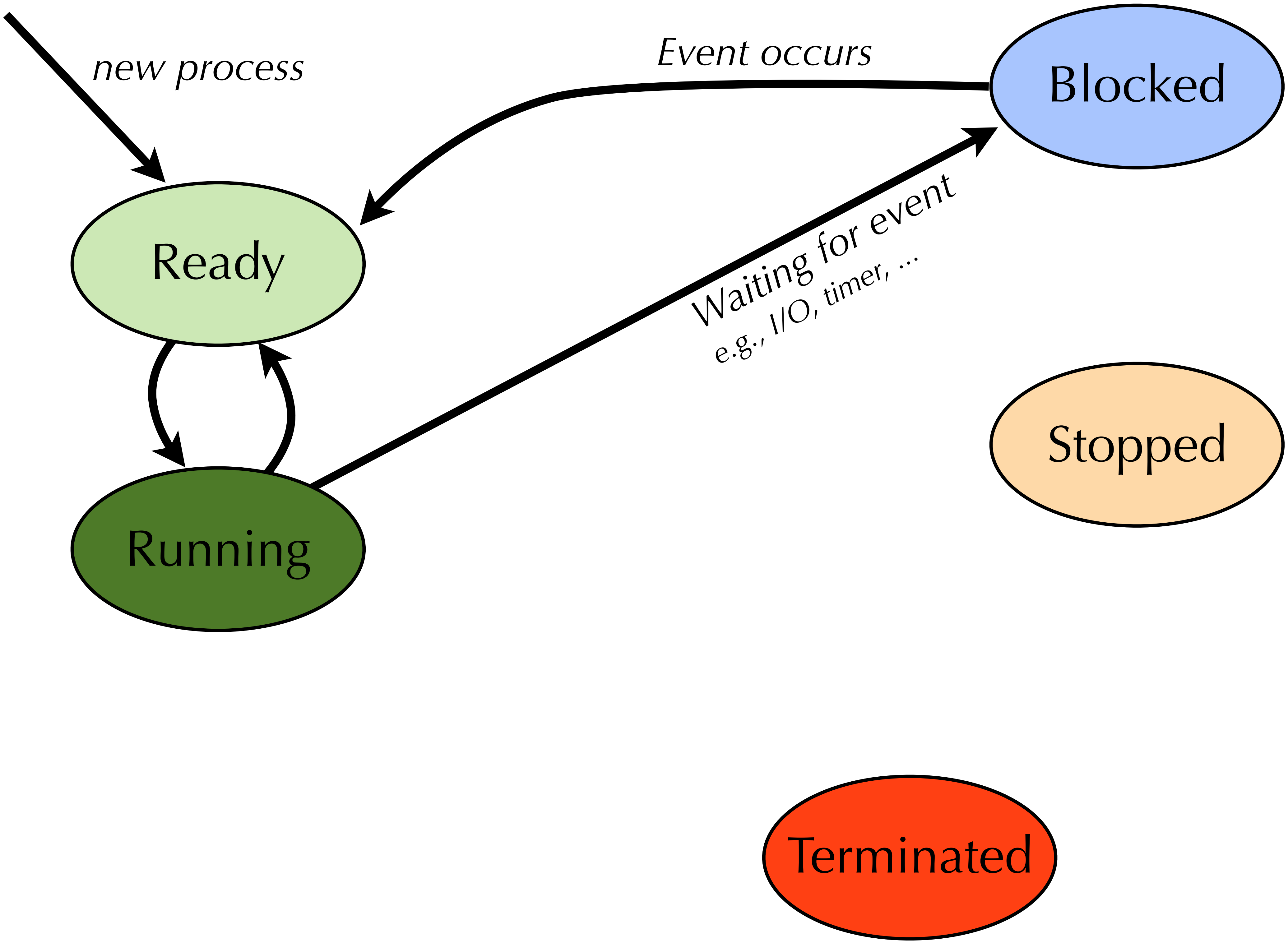
Process lifecycle



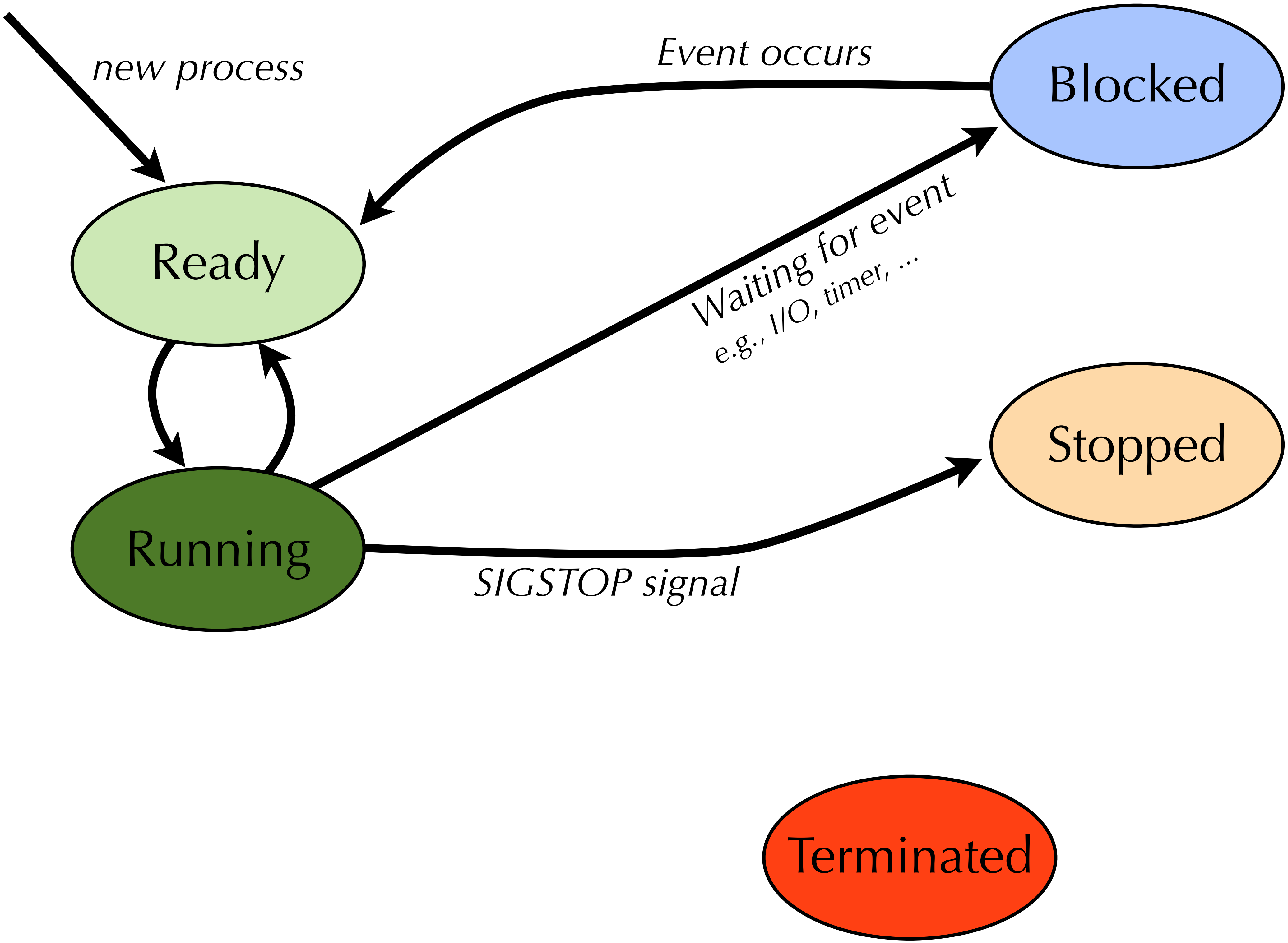
Process lifecycle



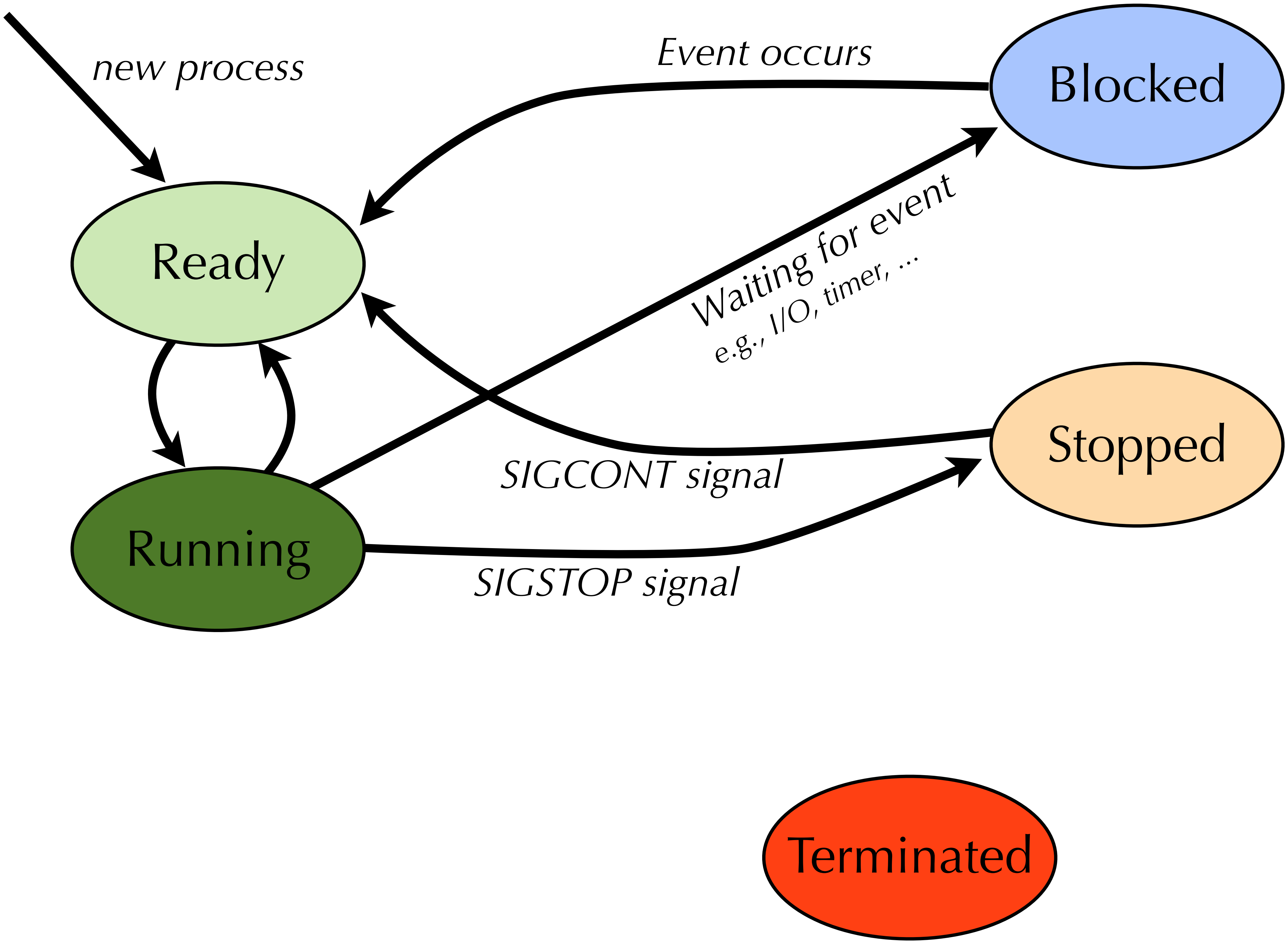
Process lifecycle



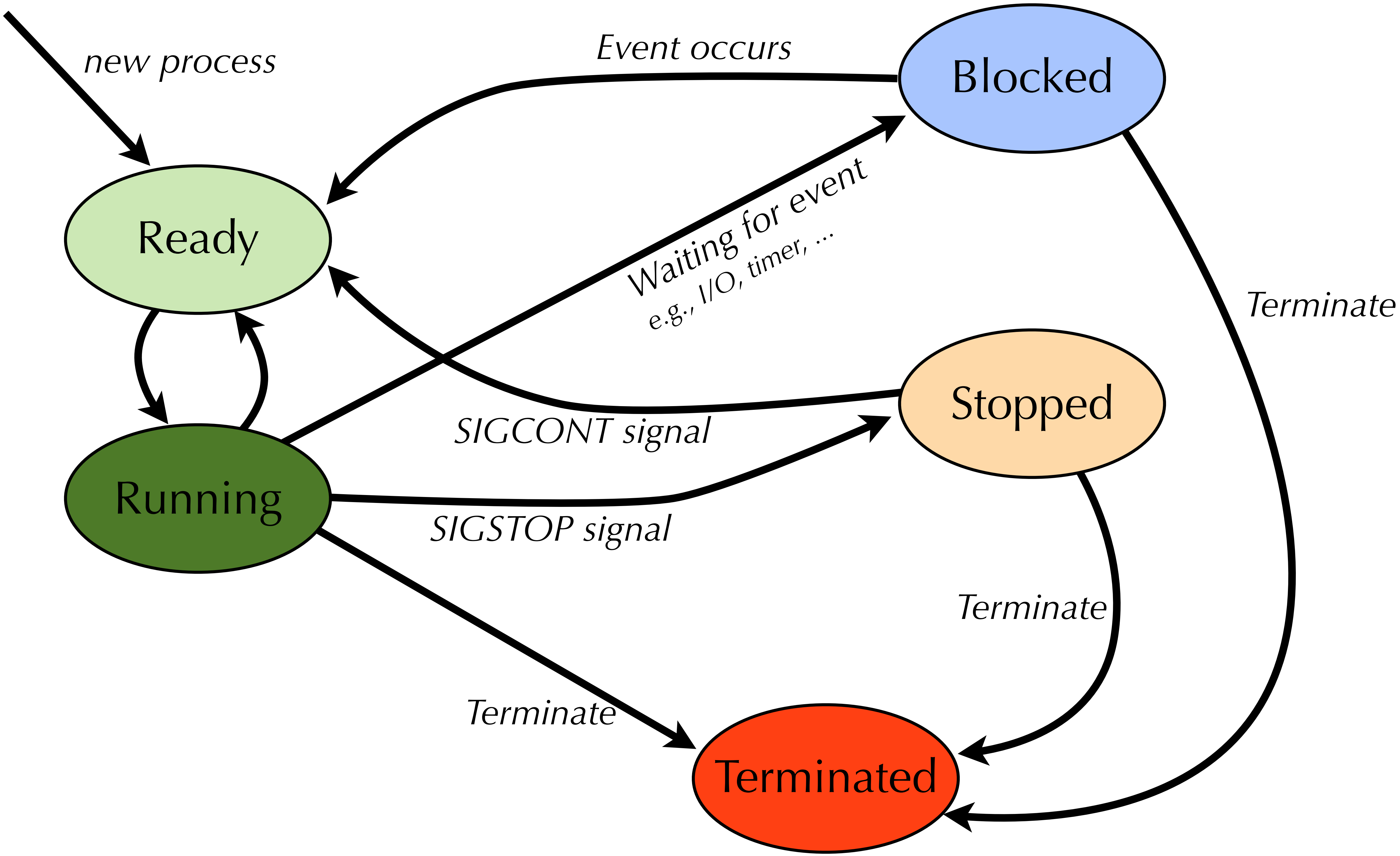
Process lifecycle



Process lifecycle



Process lifecycle



Topics for today

- The UNIX process abstraction
- Process lifecycle
  - Creating processes: forking
  - Running new programs within a process
  - Terminating and reaping processes
- Signaling processes

How are processes created?

How are processes created?

- Typically, new process is created when user runs a program
  - E.g., Double-click an application, or type a command at the shell

How are processes created?

- Typically, new process is created when user runs a program
  - E.g., Double-click an application, or type a command at the shell
- In UNIX, starting a new program is done by **some other process**
  - The shell is a process itself!
  - So are the Dock and Finder in MacOS (a variant of UNIX)

How are processes created?

- Typically, new process is created when user runs a program
  - E.g., Double-click an application, or type a command at the shell
- In UNIX, starting a new program is done by **some other process**
  - The shell is a process itself!
  - So are the Dock and Finder in MacOS (a variant of UNIX)
- One process (e.g., the shell) is creating another process (the command you want to run)
  - This is called **forking**
  - Every process has a **parent process**

How are processes created?

- Typically, new process is created when user runs a program
  - E.g., Double-click an application, or type a command at the shell
- In UNIX, starting a new program is done by **some other process**
  - The shell is a process itself!
  - So are the Dock and Finder in MacOS (a variant of UNIX)
- One process (e.g., the shell) is creating another process (the command you want to run)
  - This is called **forking**
  - Every process has a **parent process**
- Chicken-and-egg problem: How does first process get created?

How are processes created?

- Typically, new process is created when user runs a program
  - E.g., Double-click an application, or type a command at the shell
- In UNIX, starting a new program is done by **some other process**
  - The shell is a process itself!
  - So are the Dock and Finder in MacOS (a variant of UNIX)
- One process (e.g., the shell) is creating another process (the command you want to run)
  - This is called **forking**
  - Every process has a **parent process**
- Chicken-and-egg problem: How does first process get created?
  - At boot time, the OS creates the first process, called `init`, which is responsible for starting up many other processes

`fork`: Creating New Processes

`fork`: Creating New Processes

- `int fork(void)`
  - creates a new process (**child process**) that is **identical** to the calling process (**parent process**)

`fork`: Creating New Processes

- `int fork(void)`
  - creates a new process (**child process**) that is **identical** to the calling process (**parent process**)
  - returns 0 to the child process
  - returns child's process ID (pid) to the parent process

```
if (fork() == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

`fork`: Creating New Processes

- `int fork(void)`
  - creates a new process (**child process**) that is **identical** to the calling process (**parent process**)
  - returns 0 to the child process
  - returns child's process ID (pid) to the parent process

```
if (fork() == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

  - Fork is interesting (and often confusing) because it is called once but returns twice

Fork Example #1

- Parent and child process both run the same program.
  - Only difference is the return value from `fork()`
- Child's address space starts as an **exact copy** of parent's
  - They do not share the memory – instead they each have a private copy.
  - Also have the same open files with the same offsets into the files.
    - Includes stdin, stdout, and stderr

Fork Example #1

- Parent and child process both run the same program.
  - Only difference is the return value from `fork()`
- Child's address space starts as an **exact copy** of parent's
  - They do not share the memory – instead they each have a private copy.
  - Also have the same open files with the same offsets into the files.
    - Includes stdin, stdout, and stderr

```
void fork1()
{
    int x = 1;
    pid_t pid = fork();
    if (pid == 0) {
        printf("Child has x = %d\n", ++x);
    } else {
        printf("Parent has x = %d\n", --x);
    }
    printf("Bye from process %d with x = %d\n", getpid(), x);
}
```

Fork Example #1

- Parent and child process both run the same program.
  - Only difference is the return value from `fork()`
- Child's address space starts as an **exact copy** of parent's
  - They do not share the memory – instead they each have a private copy.
  - Also have the same open files with the same offsets into the files.
    - Includes stdin, stdout, and stderr

```
void fork1()
{
    int x = 1;
    pid_t pid = fork();
    if (pid == 0) {
        printf("Child has x = %d\n", ++x);
    } else {
        printf("Parent has x = %d\n", --x);
    }
    printf("Bye from process %d with x = %d\n", getpid(), x);
}
```

```
Parent has x = 0
Bye from process 9991 with x = 0
Child has x = 2
Bye from process 9992 with x = 2
```

Fork Example #2

- Key Points
  - Both parent and child can continue forking

```
void fork2()
{
    printf("L0\n");
    fork();
    printf("L1\n");
    fork();
    printf("Bye\n");
}
```

Fork Example #2

- Key Points
  - Both parent and child can continue forking

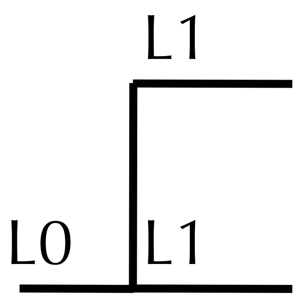
```
void fork2()
{
    printf("L0\n");
    fork();
    printf("L1\n");
    fork();
    printf("Bye\n");
}
```

L0

Fork Example #2

- Key Points
  - Both parent and child can continue forking

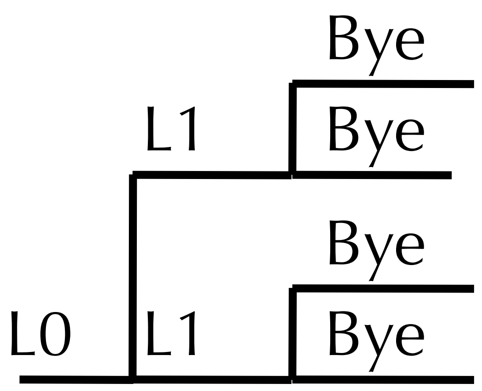
```
void fork2()
{
    printf("L0\n");
    fork();
    printf("L1\n");
    fork();
    printf("Bye\n");
}
```



Fork Example #2

- Key Points
  - Both parent and child can continue forking

```
void fork2()
{
    printf("L0\n");
    fork();
    printf("L1\n");
    fork();
    printf("Bye\n");
}
```



Fork Example #3

- Key Points
  - Both parent and child can continue forking

```
void fork3()
{
    printf("L0\n");
    fork();
    printf("L1\n");
    fork();
    printf("L2\n");
    fork();
    printf("Bye\n");
}
```

Fork Example #3

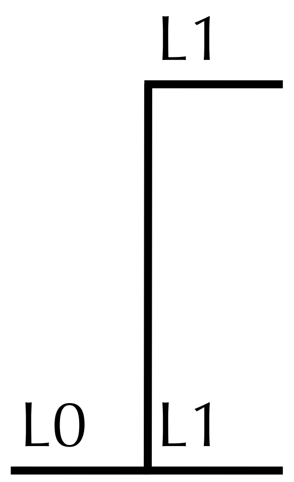
- Key Points
  - Both parent and child can continue forking

```
void fork3()
{
    printf("L0\n");
    fork();
    printf("L1\n");
    fork();
    printf("L2\n");
    fork();
    printf("Bye\n");
}
```

Fork Example #3

- Key Points
  - Both parent and child can continue forking

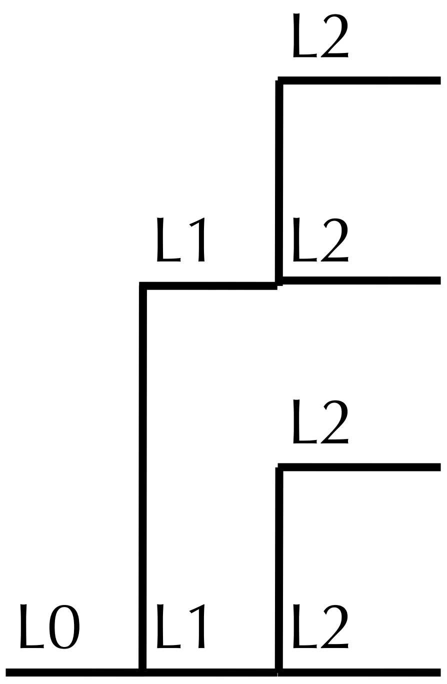
```
void fork3()
{
    printf("L0\n");
    fork();
    printf("L1\n");
    fork();
    printf("L2\n");
    fork();
    printf("Bye\n");
}
```



Fork Example #3

- Key Points
  - Both parent and child can continue forking

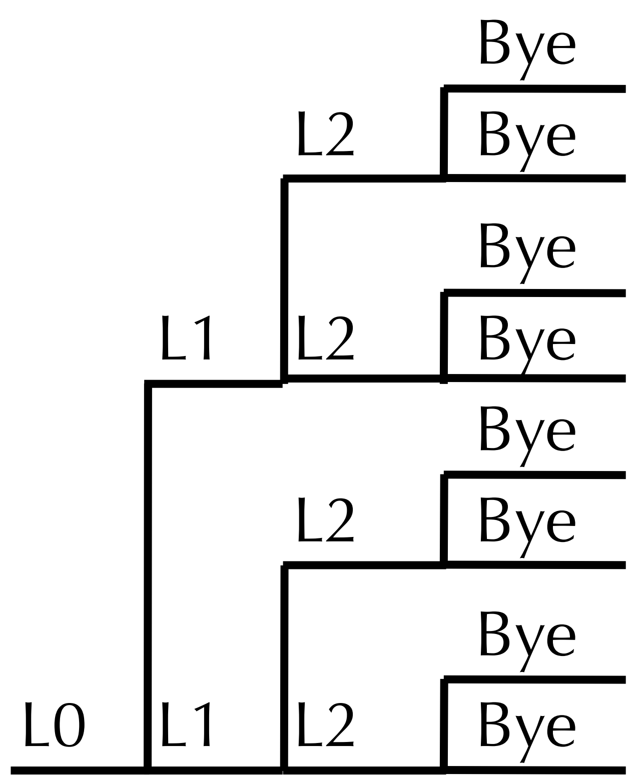
```
void fork3()
{
    printf("L0\n");
    fork();
    printf("L1\n");
    fork();
    printf("L2\n");
    fork();
    printf("Bye\n");
}
```



Fork Example #3

- Key Points
  - Both parent and child can continue forking

```
void fork3()
{
    printf("L0\n");
    fork();
    printf("L1\n");
    fork();
    printf("L2\n");
    fork();
    printf("Bye\n");
}
```



Fork Example #4

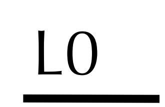
- Key Points
  - Both parent and child can continue forking

```
void fork4()
{
    printf("L0\n");
    if (fork() != 0) {
        printf("L1\n");
        if (fork() != 0) {
            printf("L2\n");
            fork();
        }
    }
    printf("Bye\n");
}
```

Fork Example #4

- Key Points
  - Both parent and child can continue forking

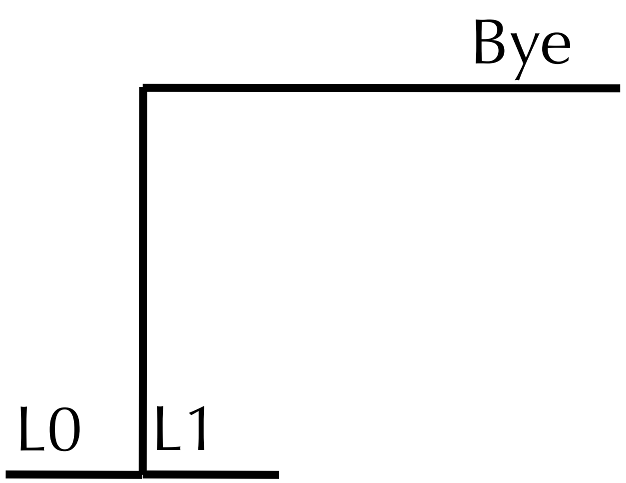
```
void fork4()
{
    printf("L0\n");
    if (fork() != 0) {
        printf("L1\n");
        if (fork() != 0) {
            printf("L2\n");
            fork();
        }
    }
    printf("Bye\n");
}
```



Fork Example #4

- Key Points
  - Both parent and child can continue forking

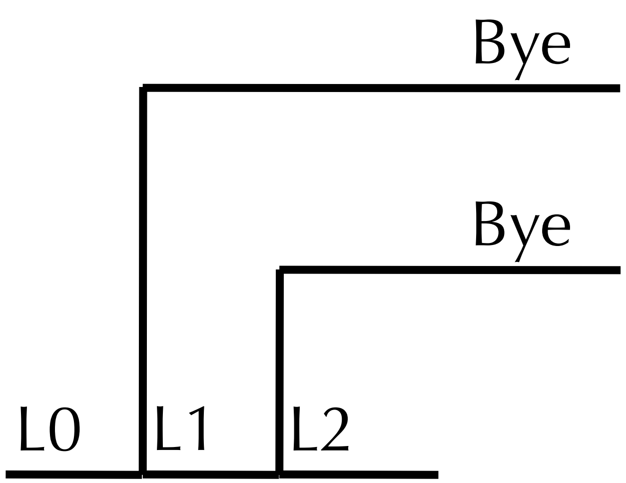
```
void fork4()
{
    printf("L0\n");
    if (fork() != 0) {
        printf("L1\n");
        if (fork() != 0) {
            printf("L2\n");
            fork();
        }
    }
    printf("Bye\n");
}
```



Fork Example #4

- Key Points
  - Both parent and child can continue forking

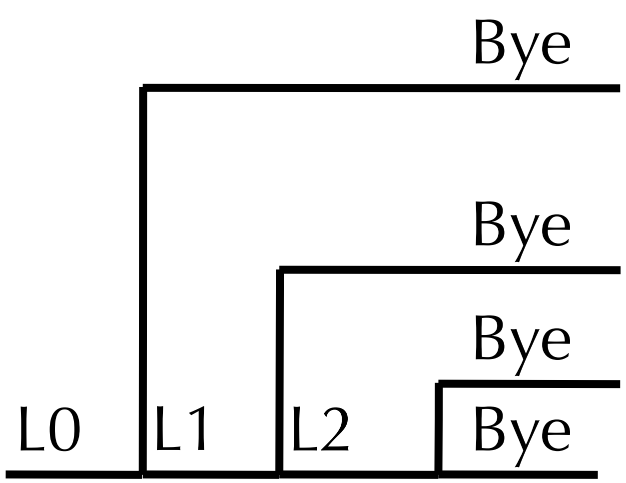
```
void fork4()
{
    printf("L0\n");
    if (fork() != 0) {
        printf("L1\n");
        if (fork() != 0) {
            printf("L2\n");
            fork();
        }
    }
    printf("Bye\n");
}
```



Fork Example #4

- Key Points
  - Both parent and child can continue forking

```
void fork4()
{
    printf("L0\n");
    if (fork() != 0) {
        printf("L1\n");
        if (fork() != 0) {
            printf("L2\n");
            fork();
        }
    }
    printf("Bye\n");
}
```



Fork Example #5

- Key Points
  - Both parent and child can continue forking

```
void fork5()
{
    printf("L0\n");
    if (fork() == 0) {
        printf("L1\n");
        if (fork() == 0) {
            printf("L2\n");
            fork();
        }
    }
    printf("Bye\n");
}
```

Fork Example #5

- Key Points
  - Both parent and child can continue forking

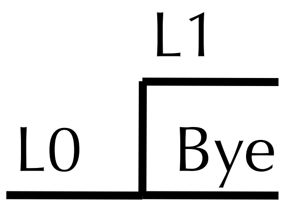
```
void fork5()
{
    printf("L0\n");
    if (fork() == 0) {
        printf("L1\n");
        if (fork() == 0) {
            printf("L2\n");
            fork();
        }
    }
    printf("Bye\n");
}
```

L0

Fork Example #5

- Key Points
  - Both parent and child can continue forking

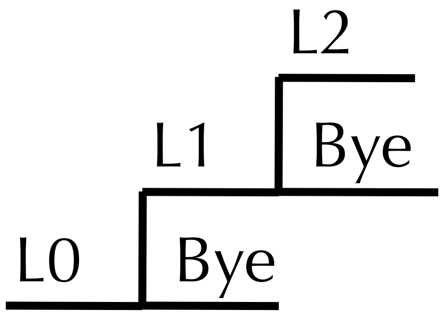
```
void fork5()
{
    printf("L0\n");
    if (fork() == 0) {
        printf("L1\n");
        if (fork() == 0) {
            printf("L2\n");
            fork();
        }
    }
    printf("Bye\n");
}
```



Fork Example #5

- Key Points
  - Both parent and child can continue forking

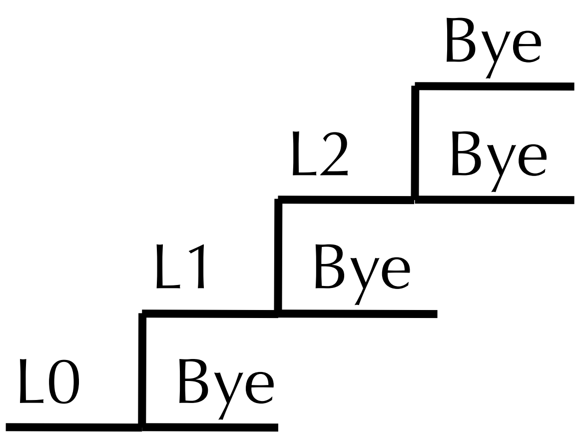
```
void fork5()
{
    printf("L0\n");
    if (fork() == 0) {
        printf("L1\n");
        if (fork() == 0) {
            printf("L2\n");
            fork();
        }
    }
    printf("Bye\n");
}
```



Fork Example #5

- Key Points
  - Both parent and child can continue forking

```
void fork5()
{
    printf("L0\n");
    if (fork() == 0) {
        printf("L1\n");
        if (fork() == 0) {
            printf("L2\n");
            fork();
        }
    }
    printf("Bye\n");
}
```



Starting new programs

Starting new programs

- How do we start a new program, instead of copying the parent?

Starting new programs

- How do we start a new program, instead of copying the parent?
- Use the UNIX `execve()` system call

Starting new programs

- How do we start a new program, instead of copying the parent?
- Use the UNIX `execve()` system call
-
```
int  execve(const  char  *filename,
                  char  *const  argv [],
                  char *const envp[]);
```
    - `filename`: name of executable file to run
    - `argv`: Command line arguments
    - `envp`: environment variable settings (e.g., $PATH, $HOME, etc.)

Starting new programs

- `execve()` does not fork a new process!
  - Rather, it replaces the address space and CPU state of the current process
  - Loads the new address space from the executable file and starts it from `main()`
  - So, to start a new program, use `fork()` followed by `execve()`

Using fork and exec

```
int main(int argc, char **argv) {
    if (fork() == 0)  {   /* Child process */
        char *newargs[3];
        printf(“Hello, I am the child process.\n”);
        newargs[0] = “/bin/echo”; /* Convention! Not required!! */
        newargs[1] = “some random string”;
        newargs[2] = NULL;      /* Indicate end of args array */
        if (execv(“/bin/echo”, newargs)) {
            printf(“warning: execv returned an error.\n”);
            exit(-1);
        }
        printf(“Child process should never get here\n”);
        exit(42);
    }
}
```

Intermission

You are a prisoner sentenced to death. The Emperor offers you a chance to live by playing a simple game. He gives you 50 black marbles, 50 white marbles and 2 empty bowls. He says, "Divide these 100 marbles into these 2 bowls, any way you like so long as you use all the marbles. Then I will blindfold you and mix the bowls around. You then can choose one bowl and remove ONE marble. If the marble is WHITE you will live, but if the marble is BLACK... you will die."

How do you divide the marbles up so that you have the greatest probability of choosing a WHITE marble?

http://www.braingle.com/

Intermission

Topics for today

- The UNIX process abstraction
- Process lifecycle
    - Creating processes: forking
    - Running new programs within a process
    - Terminating and reaping processes
- Signaling processes

Terminating a process

- A process terminates for one of 3 reasons:
  - (1) return from the `main()` procedure
  - (2) call to the `exit()` function
  - (3) receive a signal whose default action is to terminate

`exit`: Destroying Process

`exit`: Destroying Process

- `void exit(int exit_status)`

`exit`: Destroying Process

- `void exit(int exit_status)`
  - Exits a process with specified exit status.

`exit`: Destroying Process

- `void exit(int exit_status)`
  - Exits a process with specified exit status.
  - By convention, status of 0 is a "normal" exit, non-zero indicates an error of some kind.

exit: Destroying Process

- `void exit(int exit_status)`
  - Exits a process with specified exit status.
  - By convention, status of 0 is a “normal” exit, non-zero indicates an error of some kind.
  - `atexit()` registers functions to be executed upon exit.

```
void cleanup(void) {
   printf("cleaning up\n");
}

void fork6() {
   atexit(cleanup);
   fork();
   exit(0);
}
```

Zombies

Zombies

- When a process terminates (for whatever reason) OS does not remove it from system immediately

Zombies

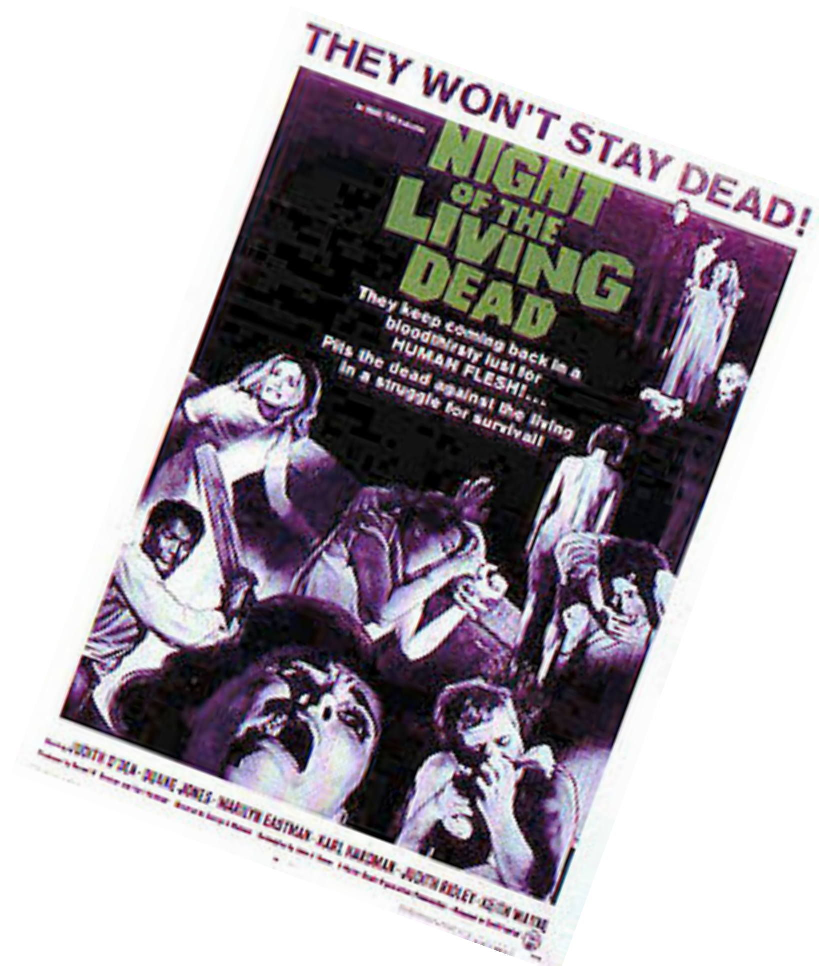
- When a process terminates (for whatever reason) OS does not remove it from system immediately
- Process stays until it is **reaped** by parent

Zombies

- When a process terminates (for whatever reason) OS does not remove it from system immediately
- Process stays until it is **reaped** by parent
    - When parent reaps a child process, OS gives the parent the exit status of child, and cleans up child

Zombies

- When a process terminates (for whatever reason) OS does not remove it from system immediately
- Process stays until it is **reaped** by parent
  - When parent reaps a child process, OS gives the parent the exit status of child, and cleans up child
  - A terminated process that has not been reaped is called a **zombie process**
- How do you reap a child process?



`wait`: Synchronizing with Children

`wait`: Synchronizing with Children

- `int wait(int *child_status)`
  - Suspends parent process until one of its children terminates
  - Return value is the pid of the child process that terminated
  - if `child_status != NULL`, it will point to the child's return status

`wait`: Synchronizing with Children

- `int wait(int *child_status)`
  - Suspends parent process until one of its children terminates
  - Return value is the pid of the child process that terminated
  - if `child_status != NULL`, it will point to the child's return status

- `child_status` can be accessed using several macros:
  - `WIFEXITED(child_status) == 1` if child exited due to call to `exit()`
  - `WEXITSTATUS(child_status)` gives the return code passed to `exit()`
  - `WCOREDUMP(child_status) == 1` if child dumped core.
  - And others (see "`man 2 wait`")

wait: Synchronizing with Children

```
void fork9() {
   int child_status;

   if (fork() == 0) {
      printf("HC: hello from child\n");
   }
   else {
      printf("HP: hello from parent\n");
      wait(&child_status);
      printf("CT: child has terminated\n");
   }
   printf("Bye\n");
   exit();
}
```

`wait`: Synchronizing with Children

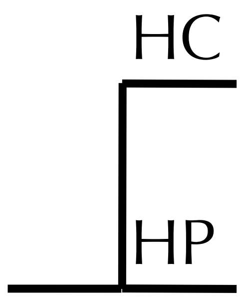
```
void fork9() {
   int child_status;

   if (fork() == 0) {
      printf("HC: hello from child\n");
   }
   else {
      printf("HP: hello from parent\n");
      wait(&child_status);
      printf("CT: child has terminated\n");
   }
   printf("Bye\n");
   exit();
}
```

`wait`: Synchronizing with Children

```
void fork9() {
   int child_status;

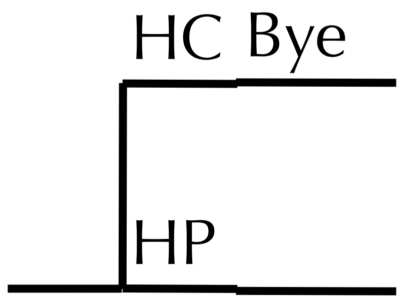
   if (fork() == 0) {
      printf("HC: hello from child\n");
   }
   else {
      printf("HP: hello from parent\n");
      wait(&child_status);
      printf("CT: child has terminated\n");
   }
   printf("Bye\n");
   exit();
}
```



wait: Synchronizing with Children

```
void fork9() {
   int child_status;

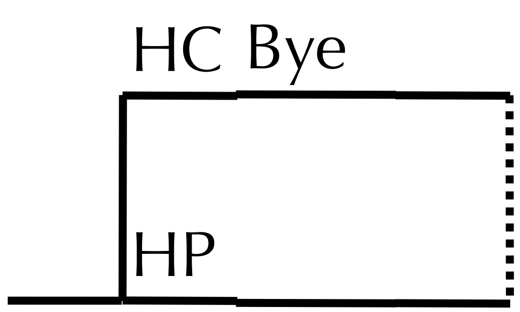
   if (fork() == 0) {
      printf("HC: hello from child\n");
   }
   else {
      printf("HP: hello from parent\n");
      wait(&child_status);
      printf("CT: child has terminated\n");
   }
   printf("Bye\n");
   exit();
}
```



`wait`: Synchronizing with Children

```
void fork9() {
   int child_status;

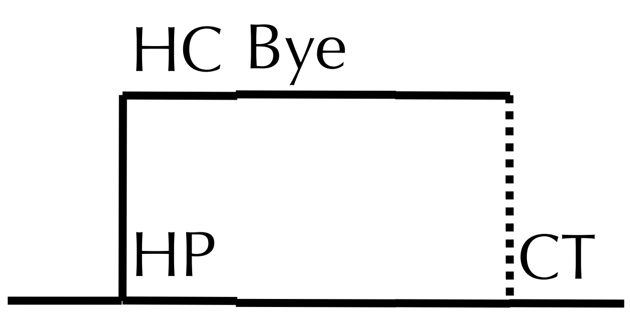
   if (fork() == 0) {
      printf("HC: hello from child\n");
   }
   else {
      printf("HP: hello from parent\n");
      wait(&child_status);
      printf("CT: child has terminated\n");
   }
   printf("Bye\n");
   exit();
}
```



wait: Synchronizing with Children

```
void fork9() {
   int child_status;

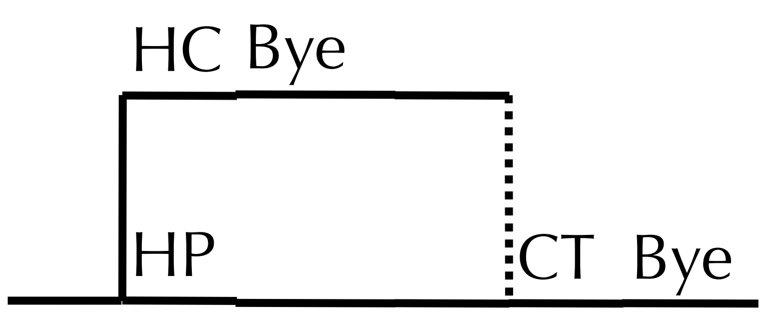
   if (fork() == 0) {
      printf("HC: hello from child\n");
   }
   else {
      printf("HP: hello from parent\n");
      wait(&child_status);
      printf("CT: child has terminated\n");
   }
   printf("Bye\n");
   exit();
}
```



wait: Synchronizing with Children

```
void fork9() {
   int child_status;

   if (fork() == 0) {
      printf("HC: hello from child\n");
   }
   else {
      printf("HP: hello from parent\n");
      wait(&child_status);
      printf("CT: child has terminated\n");
   }
   printf("Bye\n");
   exit();
}
```



What if multiple child processes exit?

- `wait()` returns status of exited children in arbitrary order.

What if multiple child processes exit?

- `wait()` returns status of exited children in arbitrary order.

```
#define N 10

void fork10()
{
    pid_t pid[N];
    int i, child_status;
    for (i = 0; i < N; i++)
        if ((pid[i] = fork()) == 0)
            exit(100+i); /* Child */
    for (i = 0; i < N; i++) {
        pid_t wpid = wait(&child_status);
        if (WIFEXITED(child_status))
            printf("Child %d terminated with exit status %d\n",
                   wpid, WEXITSTATUS(child_status));
        else
            printf("Child %d terminate abnormally\n", wpid);
    }
}
```

What if multiple child processes exit?

- `wait()` returns status of exited children in arbitrary order.

```
#define N 10

void fork10()
{
    pid_t pid[N];
    int i, child_status;
    for (i = 0; i < N; i++)
        if ((pid[i] = fork()) == 0)
            exit(100+i); /* Child */
    for (i = 0; i < N; i++) {
        pid_t wpid = wait(&child_status);
        if (WIFEXITED(child_status))
            printf("Child %d terminated with exit status %d\n",
                   wpid, WEXITSTATUS(child_status));
        else
            printf("Child %d terminate abnormally\n", wpid);
    }
}
```

```
linux> ./fork10
Child 2625 terminated with exit status 195
Child 2627 terminated with exit status 197
Child 2626 terminated with exit status 196
Child 2624 terminated with exit status 194
Child 2623 terminated with exit status 193
Child 2622 terminated with exit status 192
Child 2621 terminated with exit status 191
Child 2620 terminated with exit status 190
...
```

`waitpid()`: Waiting for a specific process

`waitpid()`: Waiting for a specific process

- ```
  pid_t waitpid(pid_t child_pid,
                int *status,
                int options)
  ```
  - Causes parent to wait for a **specific** child process to exit.
  ```

`waitpid()`: Waiting for a specific process

- ```
  pid_t waitpid(pid_t child_pid,
                int *status,
                int options)
  ```
  - Causes parent to wait for a **specific** child process to exit.
- Most general form of wait
  - `child_pid > 0`: wait for specific child to exit
  - `child_pid = -1`: wait for any child to exit
  ```

`waitpid()`: Waiting for a specific process

- ```
  pid_t waitpid(pid_t child_pid,
                int *status,
                int options)
  ```
  - Causes parent to wait for a **specific** child process to exit.
- Most general form of wait
  - `child_pid > 0`: wait for specific child to exit
  - `child_pid = -1`: wait for any child to exit
  - return value is PID of child process
  ```

`waitpid()`: Waiting for a specific process

- ```
  pid_t waitpid(pid_t child_pid,
                int *status,
                int options)
  ```
  - Causes parent to wait for a **specific** child process to exit.
- Most general form of wait
  - `child_pid > 0`: wait for specific child to exit
  - `child_pid = -1`: wait for any child to exit
  - return value is PID of child process
  - `options` can be used to specify if call should return immediately (with return value of 0) if no terminated children, and also whether we are interested in stopped processes
  ```

`waitpid()`: Waiting for a specific process

- `pid_t waitpid(pid_t child_pid,`
  `int *status,`
  `int options)`
  - Causes parent to wait for a **specific** child process to exit.
- Most general form of wait
  - `child_pid > 0`: wait for specific child to exit
  - `child_pid = -1`: wait for any child to exit
  - return value is PID of child process
  - `options` can be used to specify if call should return immediately (with return value of 0) if no terminated children, and also whether we are interested in stopped processes
  - `status` encodes information about how child exited (or was stopped)

`waitpid()`: Waiting for a specific process

- ```
  pid_t waitpid(pid_t child_pid,
                int *status,
                int options)
  ```
  ```

`waitpid()`: Waiting for a specific process

- `pid_t waitpid(pid_t child_pid,`
  `int *status,`
  `int options)`

```
void fork11()
{
    pid_t pid[N];
    int i;
    int child_status;
    for (i = 0; i < N; i++)
        if ((pid[i] = fork()) == 0)
            exit(100+i); /* Child */
    for (i = 0; i < N; i++) {
        pid_t wpid = waitpid(pid[i], &child_status, 0);
        if (WIFEXITED(child_status))
            printf("Child %d terminated with exit status %d\n",
                   wpid, WEXITSTATUS(child_status));
        else
            printf("Child %d terminated abnormally\n", wpid);
    }
```

`waitpid()`: Waiting for a specific process

- `pid_t waitpid(pid_t child_pid, int *status, int options)`

```
void fork11()
{
    pid_t pid[N];
    int i;
    int child_status;
    for (i = 0; i < N; i++)
        if ((pid[i] = fork()) == 0)
            exit(100+i); /* Child */
    for (i = 0; i < N; i++) {
        pid_t wpid = waitpid(pid[i], &child_status, 0);
        if (WIFEXITED(child_status))
            printf("Child %d terminated with exit status %d\n",
                   wpid, WEXITSTATUS(child_status));
        else
            printf("Child %d terminated abnormally\n", wpid);
    }
```

```
linux> ./fork11
Child 3064 terminated with exit status 100
Child 3065 terminated with exit status 101
Child 3066 terminated with exit status 102
Child 3067 terminated with exit status 103
Child 3068 terminated with exit status 104
Child 3069 terminated with exit status 105
Child 3070 terminated with exit status 106
...
```

Back to the zombies...

```
void zombie()
{
    if (fork() == 0) {
        /* Child */
        printf("Terminating Child, PID = %d\n",
               getpid());
        exit(0);
    } else {
        printf("Running Parent, PID = %d\n",
               getpid());
        while (1)
            ; /* Infinite loop */
    }
}
```

Back to the zombies...

- Zombie example

```
void zombie()
{
    if (fork() == 0) {
        /* Child */
        printf("Terminating Child, PID = %d\n",
               getpid());
        exit(0);
    } else {
        printf("Running Parent, PID = %d\n",
               getpid());
        while (1)
            ; /* Infinite loop */
    }
}
```

Back to the zombies...

- Zombie example

```
void zombie()
{
    if (fork() == 0) {
        /* Child */
        printf("Terminating Child, PID = %d\n",
               getpid());
        exit(0);
    } else {
        printf("Running Parent, PID = %d\n",
               getpid());
        while (1)
            ; /* Infinite loop */
    }
}
```

```
linux> ./zombie &
```

Back to the zombies...

- Zombie example

```
void zombie()
{
    if (fork() == 0) {
        /* Child */
        printf("Terminating Child, PID = %d\n",
               getpid());
        exit(0);
    } else {
        printf("Running Parent, PID = %d\n",
               getpid());
        while (1)
            ; /* Infinite loop */
    }
}
```

```
linux> ./zombie &
[1] 6639
Running Parent, PID = 6639
Terminating Child, PID = 6640
```

Back to the zombies...

- Zombie example

```
void zombie()
{
    if (fork() == 0) {
        /* Child */
        printf("Terminating Child, PID = %d\n",
               getpid());
        exit(0);
    } else {
        printf("Running Parent, PID = %d\n",
               getpid());
        while (1)
            ; /* Infinite loop */
    }
}
```

```
linux> ./zombie &
[1] 6639
Running Parent, PID = 6639
Terminating Child, PID = 6640
linux> ps
  PID TTY          TIME CMD
 6585 ttyp9    00:00:00 tcsh
 6639 ttyp9    00:00:03 zombie
 6640 ttyp9    00:00:00 zombie <defunct>
 6641 ttyp9    00:00:00 ps
linux>
```

Back to the zombies...

- Zombie example

```
void zombie()
{
    if (fork() == 0) {
        /* Child */
        printf("Terminating Child, PID = %d\n",
               getpid());
        exit(0);
    } else {
        printf("Running Parent, PID = %d\n",
               getpid());
        while (1)
            ; /* Infinite loop */
    }
}
```

```
linux> ./zombie &
[1] 6639
Running Parent, PID = 6639
Terminating Child, PID = 6640
linux> ps
  PID TTY          TIME CMD
 6585 ttyp9    00:00:00 tcsh
 6639 ttyp9    00:00:03 zombie
 6640 ttyp9    00:00:00 zombie <defunct>
 6641 ttyp9    00:00:00 ps
linux>
```

- ps shows child process as “defunct”

Orphans

Orphans

- So bad things happen if the parent does not wait for the child...

Orphans

- So bad things happen if the parent does not wait for the child…
- If the child exits first, child becomes a zombie

Orphans

- So bad things happen if the parent does not wait for the child…
- If the child exits first, child becomes a zombie
- If the parent exits first, the child becomes an **orphan**.

Orphans

- So bad things happen if the parent does not wait for the child…
- If the child exits first, child becomes a zombie
- If the parent exits first, the child becomes an **orphan**.
  - Problem: All processes (except for init) need a parent process.

Orphans

- So bad things happen if the parent does not wait for the child…
- If the child exits first, child becomes a zombie
- If the parent exits first, the child becomes an **orphan**.
  - Problem: All processes (except for init) need a parent process.
  - Orphan processes “adopted” by `init` (PID 1 on most UNIX systems)

Orphans

- So bad things happen if the parent does not wait for the child...
- If the child exits first, child becomes a zombie
- If the parent exits first, the child becomes an **orphan**.
  - Problem: All processes (except for init) need a parent process.
  - Orphan processes "adopted" by `init` (PID 1 on most UNIX systems)
  - If child subsequently terminates, it will be reaped by `init`
    - `init` reaps zombie orphans...

Nonterminating Child Example

```
void fork8()
{
    if (fork() == 0) {
        /* Child */
        printf("Running Child, PID = %d\n",
               getpid());
        while (1)
            ; /* Infinite loop */
    } else {
        printf("Terminating Parent, PID = %d\n",
               getpid());
        exit(0);
    }
}
```

- Child process still active even though parent has terminated
- Must kill explicitly, or else will keep running indefinitely

Nonterminating Child Example

```
void fork8()
{
    if (fork() == 0) {
        /* Child */
        printf("Running Child, PID = %d\n",
               getpid());
        while (1)
            ; /* Infinite loop */
    } else {
        printf("Terminating Parent, PID = %d\n",
               getpid());
        exit(0);
    }
}
```

```
linux> ./fork8
Terminating Parent, PID = 6675
Running Child, PID = 6676
```

- Child process still active even though parent has terminated
- Must kill explicitly, or else will keep running indefinitely

Nonterminating Child Example

```
void fork8()
{
    if (fork() == 0) {
        /* Child */
        printf("Running Child, PID = %d\n",
               getpid());
        while (1)
            ; /* Infinite loop */
    } else {
        printf("Terminating Parent, PID = %d\n",
               getpid());
        exit(0);
    }
}
```

```
linux> ./fork8
Terminating Parent, PID = 6675
Running Child, PID = 6676
linux> ps
  PID TTY          TIME CMD
 6585 ttyp9    00:00:00 tcsh
 6676 ttyp9    00:00:06 fork8
 6677 ttyp9    00:00:00 ps
```

- Child process still active even though parent has terminated
- Must kill explicitly, or else will keep running indefinitely

Nonterminating Child Example

```
void fork8()
{
    if (fork() == 0) {
        /* Child */
        printf("Running Child, PID = %d\n",
               getpid());
        while (1)
            ; /* Infinite loop */
    } else {
        printf("Terminating Parent, PID = %d\n",
               getpid());
        exit(0);
    }
}
```

```
linux> ./fork8
Terminating Parent, PID = 6675
Running Child, PID = 6676
linux> ps
  PID TTY          TIME CMD
 6585 ttyp9    00:00:00 tcsh
 6676 ttyp9    00:00:06 fork8
 6677 ttyp9    00:00:00 ps
linux> kill 6676
```

- Child process still active even though parent has terminated
- Must kill explicitly, or else will keep running indefinitely

Nonterminating Child Example

```
void fork8()
{
    if (fork() == 0) {
        /* Child */
        printf("Running Child, PID = %d\n",
               getpid());
        while (1)
            ; /* Infinite loop */
    } else {
        printf("Terminating Parent, PID = %d\n",
               getpid());
        exit(0);
    }
}
```

```
linux> ./fork8
Terminating Parent, PID = 6675
Running Child, PID = 6676
linux> ps
  PID TTY          TIME CMD
 6585 ttyp9    00:00:00 tcsh
 6676 ttyp9    00:00:06 fork8
 6677 ttyp9    00:00:00 ps
linux> kill 6676
linux> ps
  PID TTY          TIME CMD
 6585 ttyp9    00:00:00 tcsh
 6678 ttyp9    00:00:00 ps
```

- Child process still active even though parent has terminated
- Must kill explicitly, or else will keep running indefinitely

Zombie orphan

```
void zombie()
{
    if (fork() == 0) {
        /* Child */
        printf("Terminating Child, PID = %d\n",
               getpid());
        exit(0);
    } else {
        printf("Running Parent, PID = %d\n",
               getpid());
        while (1)
            ; /* Infinite loop */
    }
}
```

Zombie orphan

- Zombie example

```
void zombie()
{
    if (fork() == 0) {
        /* Child */
        printf("Terminating Child, PID = %d\n",
               getpid());
        exit(0);
    } else {
        printf("Running Parent, PID = %d\n",
               getpid());
        while (1)
            ; /* Infinite loop */
    }
}
```

Zombie orphan

- Zombie example

```
void zombie()
{
    if (fork() == 0) {
        /* Child */
        printf("Terminating Child, PID = %d\n",
               getpid());
        exit(0);
    } else {
        printf("Running Parent, PID = %d\n",
               getpid());
        while (1)
            ; /* Infinite loop */
    }
}
```

```
linux> ./zombie &
[1] 6639
Running Parent, PID = 6639
Terminating Child, PID = 6640
```

Zombie orphan

- Zombie example

```
void zombie()
{
    if (fork() == 0) {
        /* Child */
        printf("Terminating Child, PID = %d\n",
               getpid());
        exit(0);
    } else {
        printf("Running Parent, PID = %d\n",
               getpid());
        while (1)
            ; /* Infinite loop */
    }
}
```

```
linux> ./zombie &
[1] 6639
Running Parent, PID = 6639
Terminating Child, PID = 6640
linux> ps
  PID TTY          TIME CMD
 6585 ttyp9    00:00:00 tcsh
 6639 ttyp9    00:00:03 zombie
 6640 ttyp9    00:00:00 zombie <defunct>
 6641 ttyp9    00:00:00 ps
```

Zombie orphan

- Zombie example

```
void zombie()
{
    if (fork() == 0) {
        /* Child */
        printf("Terminating Child, PID = %d\n",
               getpid());
        exit(0);
    } else {
        printf("Running Parent, PID = %d\n",
               getpid());
        while (1)
            ; /* Infinite loop */
    }
}
```

```
linux> ./zombie &
[1] 6639
Running Parent, PID = 6639
Terminating Child, PID = 6640
linux> ps
  PID TTY          TIME CMD
 6585 ttyp9    00:00:00 tcsh
 6639 ttyp9    00:00:03 zombie
 6640 ttyp9    00:00:00 zombie <defunct>
 6641 ttyp9    00:00:00 ps
```

- ps shows child process as “defunct”

Zombie orphan

- Zombie example

```
void zombie()
{
    if (fork() == 0) {
        /* Child */
        printf("Terminating Child, PID = %d\n",
               getpid());
        exit(0);
    } else {
        printf("Running Parent, PID = %d\n",
               getpid());
        while (1)
            ; /* Infinite loop */
    }
}
```

```
linux> ./zombie &
[1] 6639
Running Parent, PID = 6639
Terminating Child, PID = 6640
linux> ps
  PID TTY          TIME CMD
 6585 ttyp9    00:00:00 tcsh
 6639 ttyp9    00:00:03 zombie
 6640 ttyp9    00:00:00 zombie <defunct>
 6641 ttyp9    00:00:00 ps
```

- ps shows child process as “defunct”
- Killing parent allows child to be reaped

Zombie orphan

- Zombie example

```
void zombie()
{
    if (fork() == 0) {
        /* Child */
        printf("Terminating Child, PID = %d\n",
               getpid());
        exit(0);
    } else {
        printf("Running Parent, PID = %d\n",
               getpid());
        while (1)
            ; /* Infinite loop */
    }
}
```

```
linux> ./zombie &
[1] 6639
Running Parent, PID = 6639
Terminating Child, PID = 6640
linux> ps
  PID TTY          TIME CMD
 6585 ttyp9    00:00:00 tcsh
 6639 ttyp9    00:00:03 zombie
 6640 ttyp9    00:00:00 zombie <defunct>
 6641 ttyp9    00:00:00 ps
linux> kill 6639
[1]    Terminated
```

- ps shows child process as "defunct"
- Killing parent allows child to be reaped

Zombie orphan

- Zombie example

```
void zombie()
{
    if (fork() == 0) {
        /* Child */
        printf("Terminating Child, PID = %d\n",
               getpid());
        exit(0);
    } else {
        printf("Running Parent, PID = %d\n",
               getpid());
        while (1)
            ; /* Infinite loop */
    }
}
```

```
linux> ./zombie &
[1] 6639
Running Parent, PID = 6639
Terminating Child, PID = 6640
linux> ps
  PID TTY          TIME CMD
 6585 ttyp9    00:00:00 tcsh
 6639 ttyp9    00:00:03 zombie
 6640 ttyp9    00:00:00 zombie <defunct>
 6641 ttyp9    00:00:00 ps
linux> kill 6639
[1]    Terminated
linux> ps
  PID TTY          TIME CMD
 6585 ttyp9    00:00:00 tcsh
 6642 ttyp9    00:00:00 ps
```

- ps shows child process as “defunct”
- Killing parent allows child to be reaped

Topics for today

- The UNIX process abstraction
- Process lifecycle
  - Creating processes: forking
  - Running new programs within a process
  - Terminating and reaping processes
- Signaling processes

Signals

- Unix provides a mechanism to allow processes and OS to interrupt other processes
- A **signal** is small message to notify a process of some system event
  - These messages not normally visible to the program
  - e.g.,

Signals

- Unix provides a mechanism to allow processes and OS to interrupt other processes
- A **signal** is small message to notify a process of some system event
  - These messages not normally visible to the program
  - e.g.,

Signals

- Unix provides a mechanism to allow processes and OS to interrupt other processes
- A **signal** is small message to notify a process of some system event
  - These messages not normally visible to the program
  - e.g.,

ID	Name	Default Action	Corresponding Event

Signals

- Unix provides a mechanism to allow processes and OS to interrupt other processes
- A **signal** is small message to notify a process of some system event
  - These messages not normally visible to the program
  - e.g.,

ID	Name	Default Action	Corresponding Event
2	SIGINT	Terminate	Interrupt (e.g., ctl-c from keyboard)

Signals

- Unix provides a mechanism to allow processes and OS to interrupt other processes
- A **signal** is small message to notify a process of some system event
  - These messages not normally visible to the program
  - e.g.,

ID	Name	Default Action	Corresponding Event
2	SIGINT	Terminate	Interrupt (e.g., ctl-c from keyboard)
9	SIGKILL	Terminate	Kill program (cannot override or ignore)

Signals

- Unix provides a mechanism to allow processes and OS to interrupt other processes
- A **signal** is small message to notify a process of some system event
  - These messages not normally visible to the program
  - e.g.,

ID	Name	Default Action	Corresponding Event
2	SIGINT	Terminate	Interrupt (e.g., ctl-c from keyboard)
9	SIGKILL	Terminate	Kill program (cannot override or ignore)
11	SIGSEGV	Terminate & Dump	Segmentation violation

Signals

- Unix provides a mechanism to allow processes and OS to interrupt other processes
- A **signal** is small message to notify a process of some system event
  - These messages not normally visible to the program
  - e.g.,

ID	Name	Default Action	Corresponding Event
2	SIGINT	Terminate	Interrupt (e.g., ctl-c from keyboard)
9	SIGKILL	Terminate	Kill program (cannot override or ignore)
11	SIGSEGV	Terminate & Dump	Segmentation violation
14	SIGALRM	Terminate	Timer signal

Signals

- Unix provides a mechanism to allow processes and OS to interrupt other processes
- A **signal** is small message to notify a process of some system event
  - These messages not normally visible to the program
  - e.g.,

ID	Name	Default Action	Corresponding Event
2	SIGINT	Terminate	Interrupt (e.g., ctl-c from keyboard)
9	SIGKILL	Terminate	Kill program (cannot override or ignore)
11	SIGSEGV	Terminate & Dump	Segmentation violation
14	SIGALRM	Terminate	Timer signal
17	SIGCHLD	Ignore	Child stopped or terminated

Signals

- Unix provides a mechanism to allow processes and OS to interrupt other processes
- A **signal** is small message to notify a process of some system event
  - These messages not normally visible to the program
  - e.g.,

ID	Name	Default Action	Corresponding Event
2	SIGINT	Terminate	Interrupt (e.g., ctl-c from keyboard)
9	SIGKILL	Terminate	Kill program (cannot override or ignore)
11	SIGSEGV	Terminate & Dump	Segmentation violation
14	SIGALRM	Terminate	Timer signal
17	SIGCHLD	Ignore	Child stopped or terminated
19	SIGSTOP	Stops process	Process asked to stop

Signals

- Unix provides a mechanism to allow processes and OS to interrupt other processes
- A **signal** is small message to notify a process of some system event
  - These messages not normally visible to the program
  - e.g.,

ID	Name	Default Action	Corresponding Event
2	SIGINT	Terminate	Interrupt (e.g., ctl-c from keyboard)
9	SIGKILL	Terminate	Kill program (cannot override or ignore)
11	SIGSEGV	Terminate & Dump	Segmentation violation
14	SIGALRM	Terminate	Timer signal
17	SIGCHLD	Ignore	Child stopped or terminated
19	SIGSTOP	Stops process	Process asked to stop
18	SIGCONT	Continue process	Process asked to continue

Signals

- Unix provides a mechanism to allow processes and OS to interrupt other processes
- A **signal** is small message to notify a process of some system event
  - These messages not normally visible to the program
  - e.g.,

ID	Name	Default Action	Corresponding Event
2	SIGINT	Terminate	Interrupt (e.g., ctl-c from keyboard)
9	SIGKILL	Terminate	Kill program (cannot override or ignore)
11	SIGSEGV	Terminate & Dump	Segmentation violation
14	SIGALRM	Terminate	Timer signal
17	SIGCHLD	Ignore	Child stopped or terminated
19	SIGSTOP	Stops process	Process asked to stop
18	SIGCONT	Continue process	Process asked to continue

Constant values may vary between platforms!

Signal concepts

Signal concepts

- Two distinct steps to transfer a signal:

Signal concepts

- Two distinct steps to transfer a signal:
  - (1) OS **sends** (**delivers**) signal to destination process

Signal concepts

- Two distinct steps to transfer a signal:
  - (1) OS **sends** (**delivers**) signal to destination process
    - either because of some system event, or because explicitly requested via `kill` function

Signal concepts

- Two distinct steps to transfer a signal:
  - (1) OS **sends** (**delivers**) signal to destination process
    - either because of some system event, or because explicitly requested via `kill` function
  - (2) Process **receives** signal (i.e., forced by OS to react to signal in some way)

Signal concepts

- Two distinct steps to transfer a signal:
  - (1) OS **sends** (**delivers**) signal to destination process
    - either because of some system event, or because explicitly requested via `kill` function
  - (2) Process **receives** signal (i.e., forced by OS to react to signal in some way)
    - Process can react in one of three ways:

Signal concepts

- Two distinct steps to transfer a signal:
  - (1) OS **sends** (**delivers**) signal to destination process
    - either because of some system event, or because explicitly requested via `kill` function
  - (2) Process **receives** signal (i.e., forced by OS to react to signal in some way)
    - Process can react in one of three ways:
      - ignore signal (i.e., do nothing)

Signal concepts

- Two distinct steps to transfer a signal:
  - (1) OS **sends** (**delivers**) signal to destination process
    - either because of some system event, or because explicitly requested via `kill` function
  - (2) Process **receives** signal (i.e., forced by OS to react to signal in some way)
    - Process can react in one of three ways:
      - ignore signal (i.e., do nothing)
      - terminate (maybe dumping core)

Signal concepts

- Two distinct steps to transfer a signal:
  - (1) OS **sends** (**delivers**) signal to destination process
    - either because of some system event, or because explicitly requested via `kill` function
  - (2) Process **receives** signal (i.e., forced by OS to react to signal in some way)
    - Process can react in one of three ways:
      - ignore signal (i.e., do nothing)
      - terminate (maybe dumping core)
      - **catch** a signal with a **signal handler** function

Signal concepts

Signal concepts

- Signal sent but not yet received is **pending**
  - At most one signal of each type is pending
  - Signals are not queued!
    - If process has pending signal of type *k*, then subsequent signals of type *k* are discarded

Signal concepts

- Signal sent but not yet received is **pending**
  - At most one signal of each type is pending
  - Signals are not queued!
    - If process has pending signal of type *k*, then subsequent signals of type *k* are discarded
- Process can **block** receipt of certain signals.
  - Blocked signals will be pending until process unblocks

Signal concepts

- Signal sent but not yet received is **pending**
  - At most one signal of each type is pending
  - Signals are not queued!
    - If process has pending signal of type *k*, then subsequent signals of type *k* are discarded
- Process can **block** receipt of certain signals.
  - Blocked signals will be pending until process unblocks
- Any signal received at most once

Pending and blocking signals

Pending and blocking signals

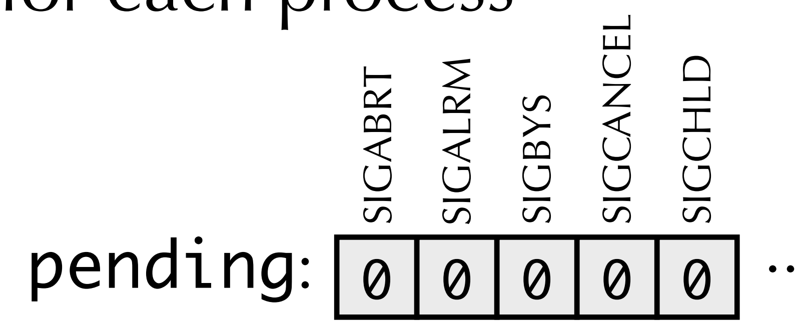
- OS maintains `pending` and `blocked` bit vectors for each process

Pending and blocking signals

- OS maintains `pending` and `blocked` bit vectors for each process
  - `pending` represents set of pending signals
    - OS sets bit *k* of pending when signal of type *k* is delivered
    - OS clears bit *k* of pending when signal of type *k* is received

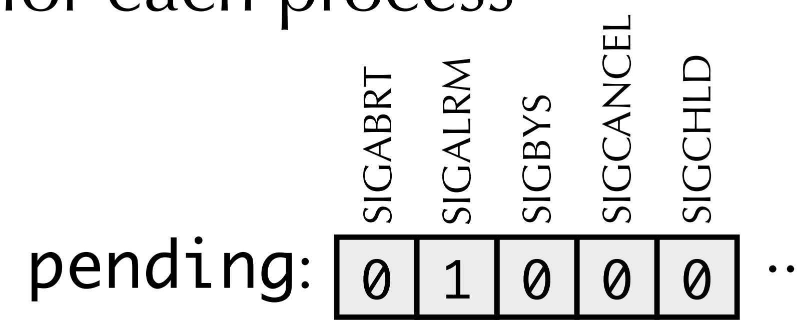
Pending and blocking signals

- OS maintains `pending` and `blocked` bit vectors for each process
  - `pending` represents set of pending signals
    - OS sets bit *k* of pending when signal of type *k* is delivered
    - OS clears bit *k* of pending when signal of type *k* is received



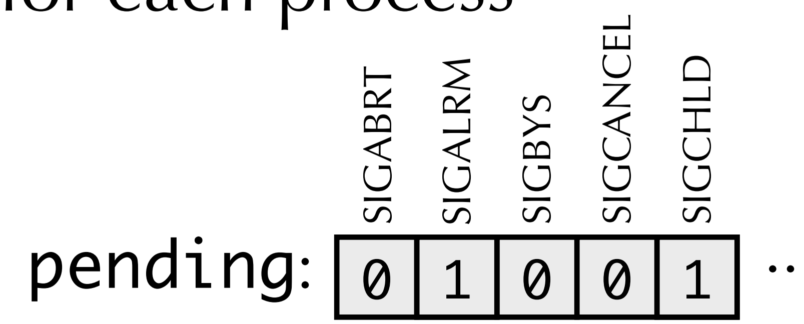
Pending and blocking signals

- OS maintains `pending` and `blocked` bit vectors for each process
  - `pending` represents set of pending signals
    - OS sets bit *k* of pending when signal of type *k* is delivered
    - OS clears bit *k* of pending when signal of type *k* is received



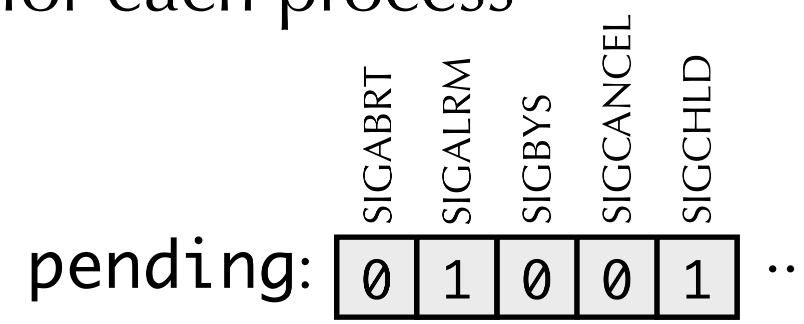
Pending and blocking signals

- OS maintains `pending` and `blocked` bit vectors for each process
  - `pending` represents set of pending signals
    - OS sets bit *k* of pending when signal of type *k* is delivered
    - OS clears bit *k* of pending when signal of type *k* is received



Pending and blocking signals

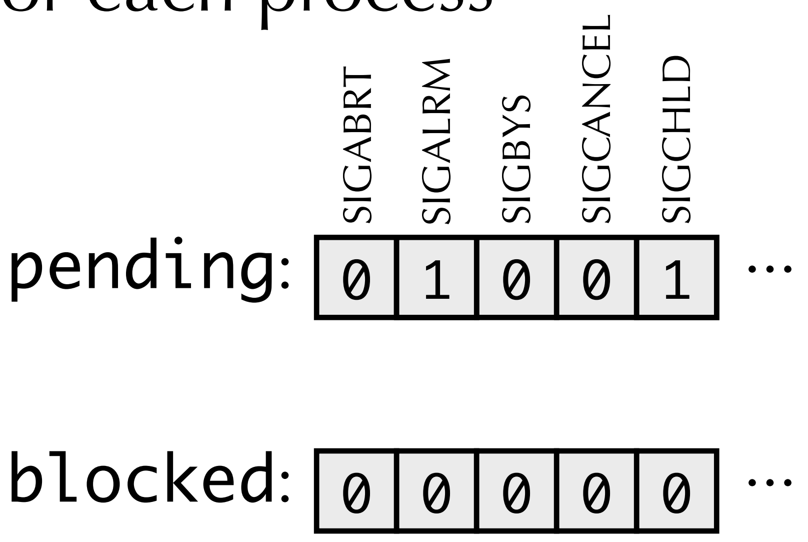
- OS maintains `pending` and `blocked` bit vectors for each process
  - `pending` represents set of pending signals
    - OS sets bit *k* of pending when signal of type *k* is delivered
    - OS clears bit *k* of pending when signal of type *k* is received



  - `blocked` represents set of signals process has blocked
    - Can be set and cleared using `sigprocmask` function

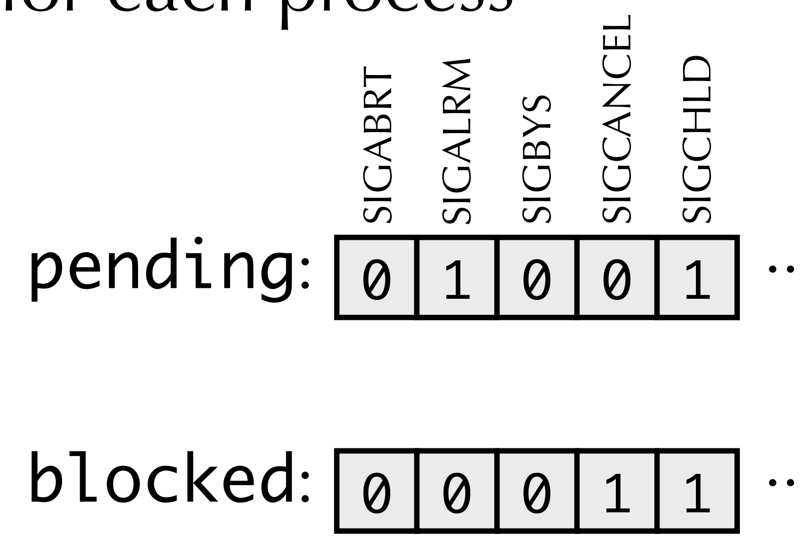
Pending and blocking signals

- OS maintains `pending` and `blocked` bit vectors for each process
  - `pending` represents set of pending signals
    - OS sets bit *k* of pending when signal of type *k* is delivered
    - OS clears bit *k* of pending when signal of type *k* is received
  - `blocked` represents set of signals process has blocked
    - Can be set and cleared using `sigprocmask` function



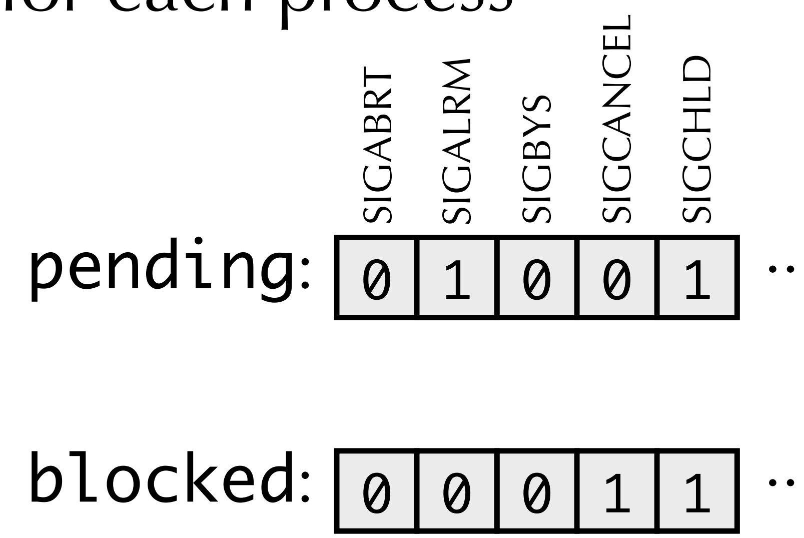
Pending and blocking signals

- OS maintains `pending` and `blocked` bit vectors for each process
  - `pending` represents set of pending signals
    - OS sets bit *k* of pending when signal of type *k* is delivered
    - OS clears bit *k* of pending when signal of type *k* is received
  - `blocked` represents set of signals process has blocked
    - Can be set and cleared using `sigprocmask` function



Pending and blocking signals

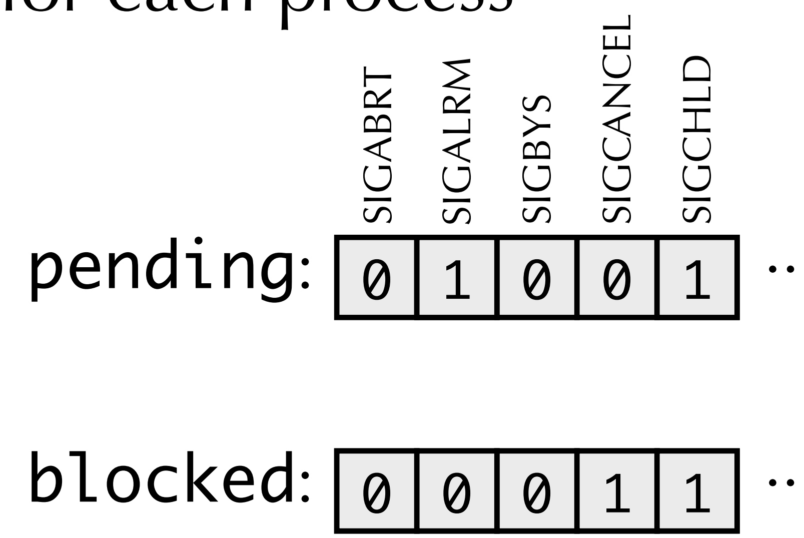
- OS maintains `pending` and `blocked` bit vectors for each process
  - `pending` represents set of pending signals
    - OS sets bit *k* of pending when signal of type *k* is delivered
    - OS clears bit *k* of pending when signal of type *k* is received
  - `blocked` represents set of signals process has blocked
    - Can be set and cleared using `sigprocmask` function



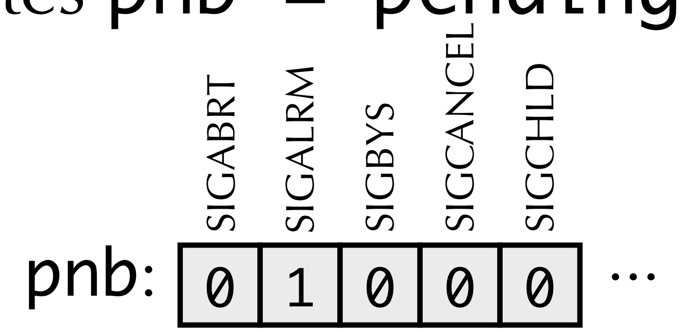
- For a process, OS computes `pnb = pending & ~blocked`

Pending and blocking signals

- OS maintains `pending` and `blocked` bit vectors for each process
  - `pending` represents set of pending signals
    - OS sets bit *k* of pending when signal of type *k* is delivered
    - OS clears bit *k* of pending when signal of type *k* is received
  - `blocked` represents set of signals process has blocked
    - Can be set and cleared using `sigprocmask` function

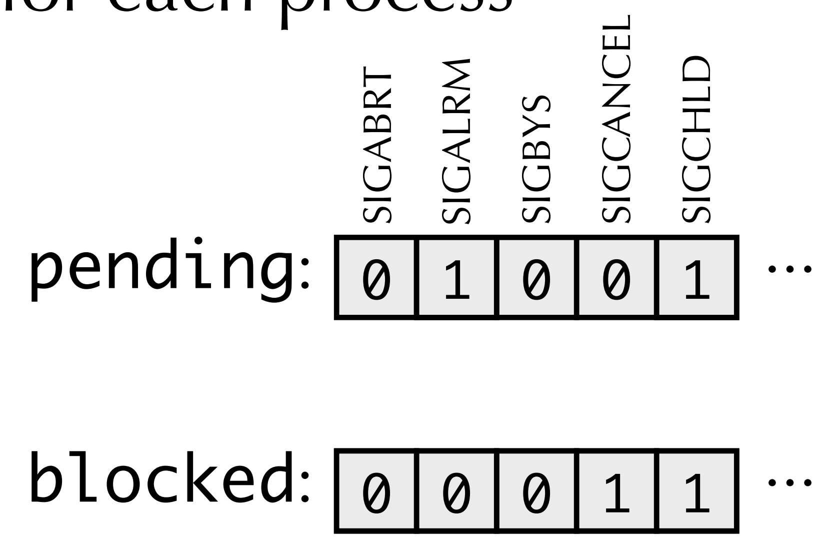


- For a process, OS computes `pnb = pending & ~blocked`

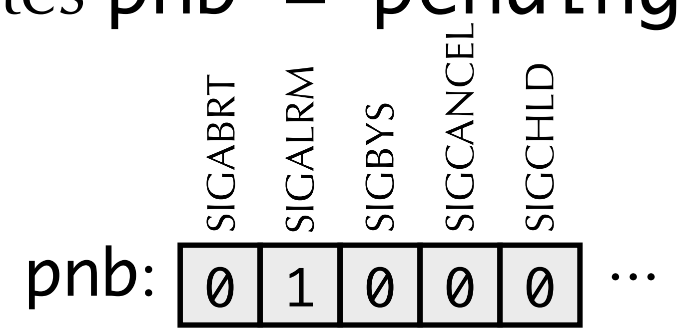


Pending and blocking signals

- OS maintains `pending` and `blocked` bit vectors for each process
  - `pending` represents set of pending signals
    - OS sets bit *k* of pending when signal of type *k* is delivered
    - OS clears bit *k* of pending when signal of type *k* is received
  - `blocked` represents set of signals process has blocked
    - Can be set and cleared using `sigprocmask` function



- For a process, OS computes `pnb = pending & ~blocked`



  - If `pnb == 0` then no signals to be received

Pending and blocking signals

- OS maintains `pending` and `blocked` bit vectors for each process
  - `pending` represents set of pending signals
    - OS sets bit *k* of pending when signal of type *k* is delivered
    - OS clears bit *k* of pending when signal of type *k* is received
  - `blocked` represents set of signals process has blocked
    - Can be set and cleared using `sigprocmask` function

	SIGABRT	SIGALRM	SIGBYS	SIGCANCEL	SIGCHLD	
`pending`:	0	1	0	0	1	…
`blocked`:	0	0	0	1	1	…

- For a process, OS computes `pnb = pending & ~blocked`

	SIGABRT	SIGALRM	SIGBYS	SIGCANCEL	SIGCHLD	
`pnb`:	0	1	0	0	0	…

  - If `pnb == 0` then no signals to be received
  - If `pnb != 0` then OS chooses a signal to be received, and triggers some action by process

Sending signals with kill

- kill programs sends an arbitrary signal to a process
  - E.g., `kill -9 24818` sends SIGKILL to process 24818
- Also a function: `kill(pid_t p, int signal)`

- Can send a signal to a specific process, or all processes in a **process group**
  - Every process belongs to a process group
  - Read textbook for more info

Default actions

- Each signal type has a predefined **default action**
- One of
  - The process terminates
  - The process terminates and dumps core
  - The process stops (until restarted by a SIGCONT signal)
  - The process ignores the action

Signal handlers

- `signal(int signum, handler_t *handler)`
  - Overrides default action for signals of kind `signum`
- Different values for `handler`
  - SIG_IGN: ignore signals of type `signum`
  - SIG_DFL: revert to the default action for signals of type `signum`
  - Otherwise, it is a function pointer for a **signal handler**
    - Function will be called on receipt of signal of type `signum`
    - Referred to as **installing** handler
    - Handler execution is called **handling** or **catching** signal
    - When handler returns, control flow of interrupted process continues

Signal handler example

```
void int_handler(int sig) {
    printf("Process %d received signal %d\n",
            getpid(), sig);
}

int main() {
    signal(SIGINT, int_handler);
    while (1)
        ;
}
```

Signal handler example

```
void int_handler(int sig) {
    printf("Process %d received signal %d\n",
            getpid(), sig);
}

int main() {
    signal(SIGINT, int_handler);
    while (1)
        ;
}
```

```
$ ./signaleg
```

Signal handler example

```
void int_handler(int sig) {
    printf("Process %d received signal %d\n",
            getpid(), sig);
}

int main() {
    signal(SIGINT, int_handler);
    while (1)
        ;
}
```

```
$ ./signaleg
^C
```

Signal handler example

```
void int_handler(int sig) {
    printf("Process %d received signal %d\n",
            getpid(), sig);
}

int main() {
    signal(SIGINT, int_handler);
    while (1)
        ;
}
```

```
$ ./signaleg
^C
Process 319 received signal 2
```

Signal handler example

```c
void int_handler(int sig) {
    printf("Process %d received signal %d\n",
            getpid(), sig);
}

int main() {
    signal(SIGINT, int_handler);
    while (1)
        ;
}
```

```
$ ./signaleg
^C
Process 319 received signal 2
^C
```

Signal handler example

```
void int_handler(int sig) {
    printf("Process %d received signal %d\n",
            getpid(), sig);
}

int main() {
    signal(SIGINT, int_handler);
    while (1)
        ;
}
```

```
$ ./signaleg
^C
Process 319 received signal 2
^C
Process 319 received signal 2
```

Signal handler example

```
void int_handler(int sig) {
    printf("Process %d received signal %d\n",
            getpid(), sig);
}

int main() {
    signal(SIGINT, int_handler);
    while (1)
        ;
}
```

```
$ ./signaleg
^C
Process 319 received signal 2
^C
Process 319 received signal 2
^C
```

Signal handler example

```
void int_handler(int sig) {
    printf("Process %d received signal %d\n",
                getpid(), sig);
}

int main() {
    signal(SIGINT, int_handler);
    while (1)
        ;
}
```

```
$ ./signaleg
^C
Process 319 received signal 2
^C
Process 319 received signal 2
^C
Process 319 received signal 2
```

Signal handler example

```
void int_handler(int sig) {
    printf("Process %d received signal %d\n",
            getpid(), sig);
}

int main() {
    signal(SIGINT, int_handler);
    while (1)
        ;
}
```

```
$ ./signaleg
^C
Process 319 received signal 2
^C
Process 319 received signal 2
^C
Process 319 received signal 2
```

```
$ kill -9 319
```

Signal handler example

```c
void int_handler(int sig) {
    printf("Process %d received signal %d\n",
            getpid(), sig);
}

int main() {
    signal(SIGINT, int_handler);
    while (1)
        ;
}
```

```
$ ./signaleg
^C
Process 319 received signal 2
^C
Process 319 received signal 2
^C
Process 319 received signal 2
Killed
```

```
$ kill -9 319
```

Signal handlers as concurrent flows

- Signal handlers run **concurrently** with main program
  - Signal handler is not a separate process
  - Concurrent here means "non-sequential", as opposed to "parallel"