香港中文大學(深圳)
The Chinese University of Hong Kong, Shenzhen

Operating Systems
Lecture 4

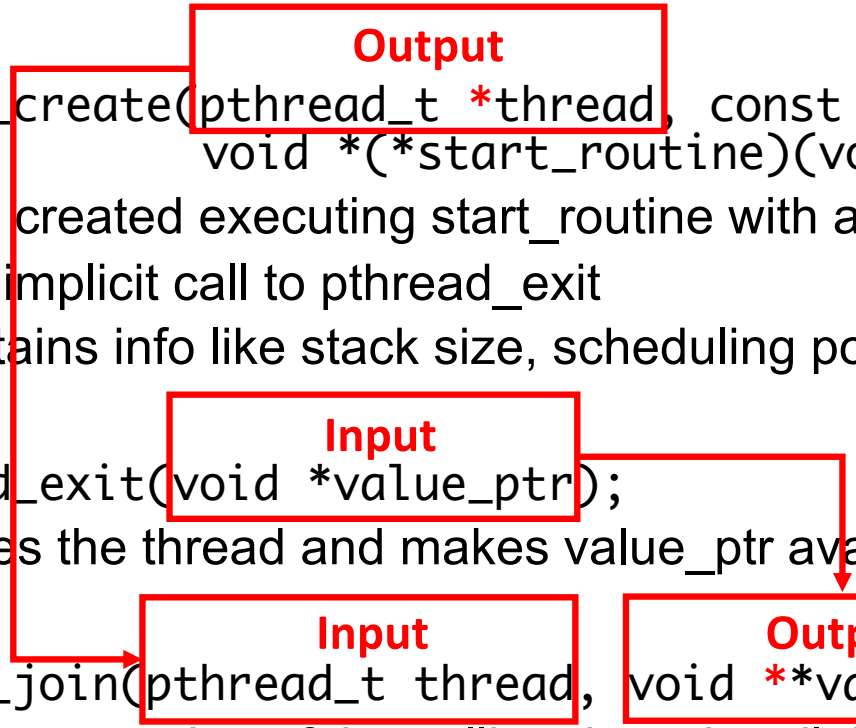
System programming: processes

Prof. Yunming XIAO

School of Data Science

Acknowledgments: Ion Stoica and Matei Zaharia, Berkeley CS 162

Recall: OS library API for threads: *pthread*



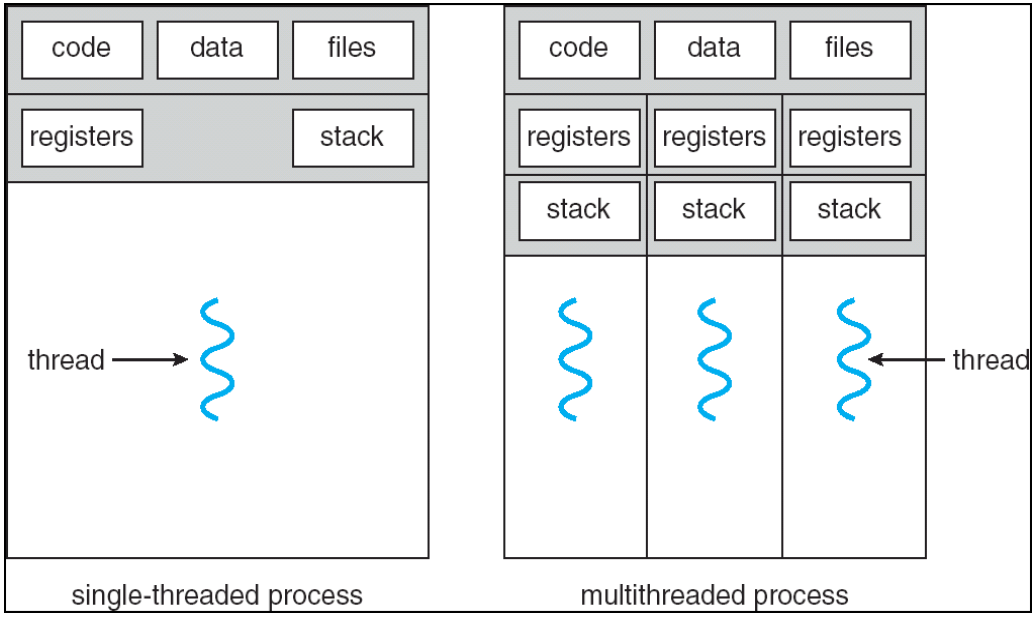
- thread is created executing start_routine with arg as its sole argument.
- return is implicit call to pthread_exit
- (attr contains info like stack size, scheduling policy, etc)

- terminates the thread and makes value_ptr available to any successful join

- suspends execution of the calling thread until the target thread terminates.
- On return with a non-NULL value_ptr the value passed to pthread_exit() by the terminating thread is made available in the location referenced by value_ptr.

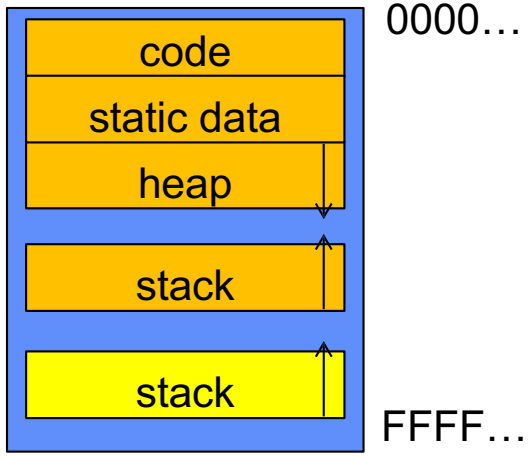
Recall: process

- Processes consists of
  - One or more threads
  - Address space, and other resources shared by the threads within the same process



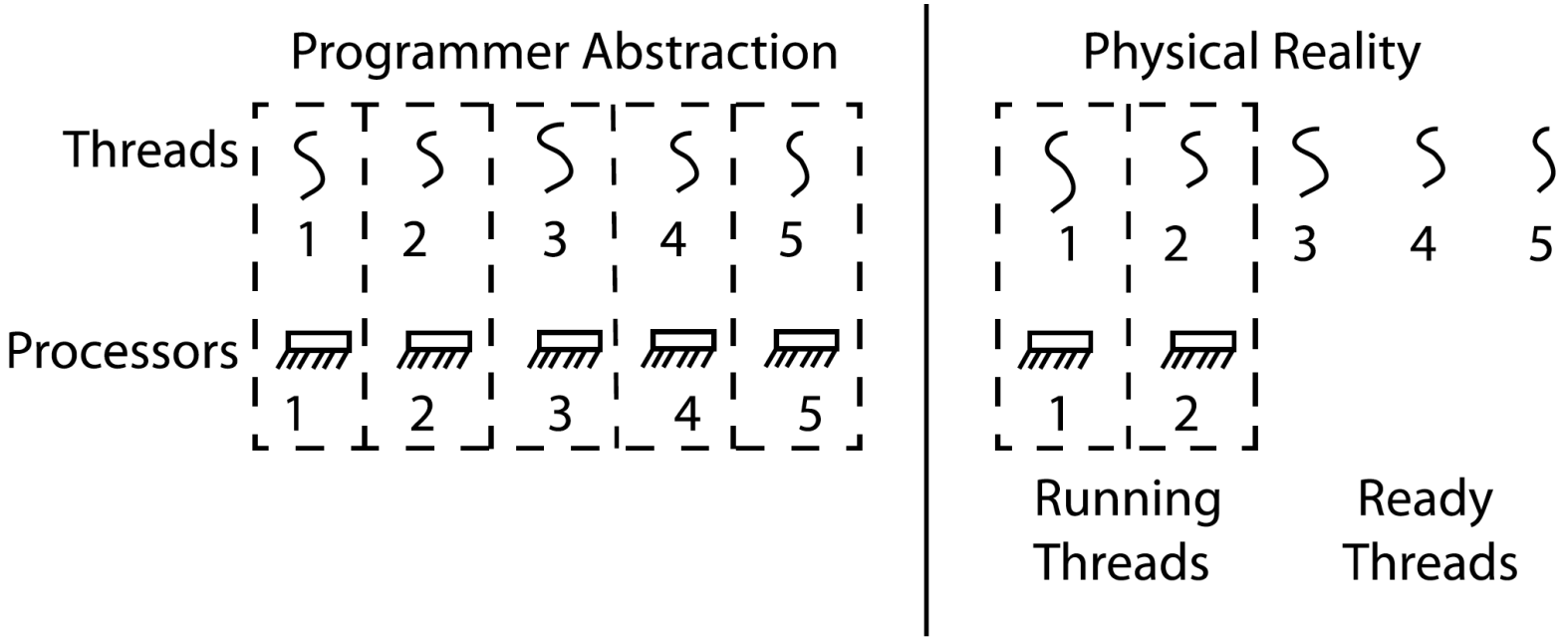
Recall: memory layout with two threads

- Two sets of CPU registers
- Two sets of Stacks
- Issues:
  - How do we position stacks relative to each other?
  - What maximum size should we choose for the stacks?
  - What happens if threads violate this?
  - How might you catch violations?



INTERLEAVING and NONEDETERMINISM
(the beginning of a long discussion)

Thread abstraction

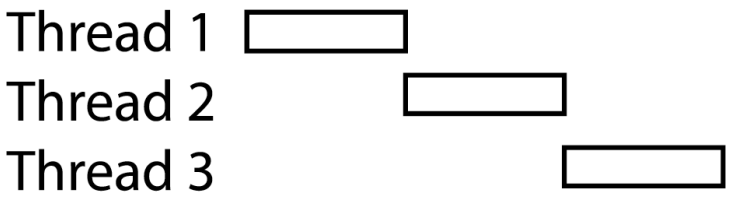


- Illusion: Infinite number of processors
- Reality: threads execute with variable “speed”
  - Programs must be designed to work with any schedule

Programmer's vs. processor's view

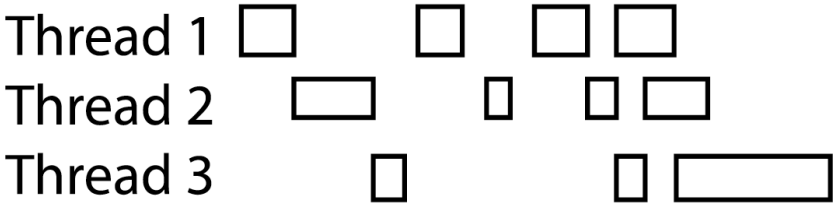
Programmer's View	Possible Execution #1	Possible Execution #2	Possible Execution #3
.	.	.	.
.	.	.	.
.	.	.	.
x = x + 1;	x = x + 1;	x = x + 1	x = x + 1
y = y + x;	y = y + x;	.....	y = y + x
z = x +5y;	z = x + 5y;	thread is suspended	.....
.	.	other thread(s) run	thread is suspended
.	.	thread is resumed	other thread(s) run
.	.	.....	thread is resumed
		y = y + x	.....
		z = x + 5y	z = x + 5y

Possible executions



a) One execution

b) Another execution



c) Another execution

Correctness vs. concurrent threads

- Non-determinism:
  - Scheduler can run threads in any order
  - Scheduler can switch threads at any time
  - This can make testing very difficult
- Independent threads
  - No state shared with other threads
  - Deterministic, reproducible conditions
- Cooperating threads
  - Shared state between multiple threads
- Goal: correctness by design

Race condition: example 1

- Initially x = 0 and y = 0

Thread A	**Thread B**
`x = 1;`	`y = 2;`

- What are the possible values of x below after all threads finish?
  - Must be 1. Thread B does not interfere

Race condition: example 2

- Initially x = 0 and y = 0

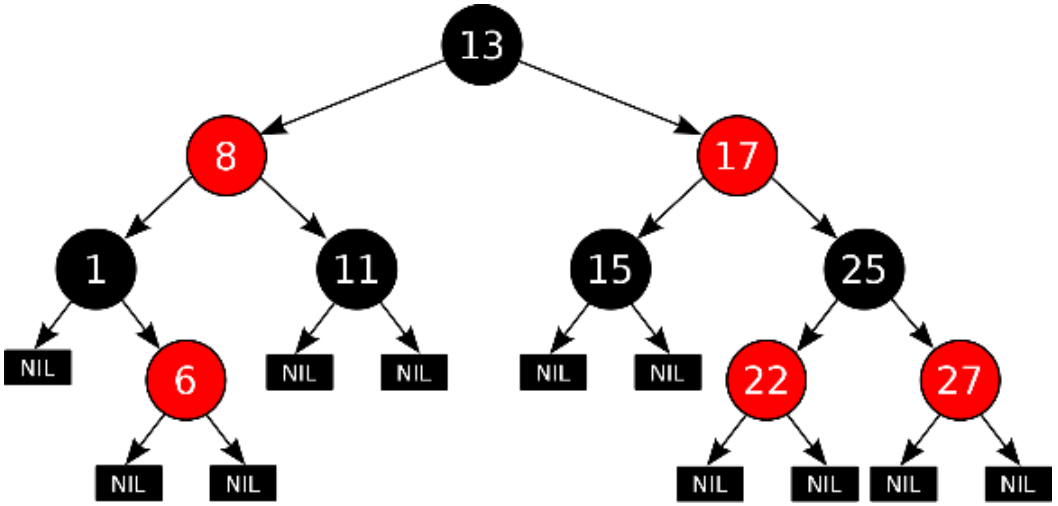
Thread A	**Thread B**
`x = y + 1;`	`y = 2;`
	`y = y * 2;`

- What are the possible values of x below?
  - 1 or 3 or 5 (non-deterministically)
- Race condition: Thread A races against Thread B!

Example: shared data structure

Thread A

```
Insert(3)
```



Tree-Based Set Data Structure

Thread B

```
Insert(4)
Get(6)
```

- How do we make sure this executes correctly?

Relevant definitions

- Synchronization: Coordination among threads, usually regarding shared data

- Mutual Exclusion: Ensuring only one thread does a particular thing at a time (one thread excludes the others)
  - Type of synchronization

- Critical Section: Code exactly one thread can execute at once
  - Result of mutual exclusion

- Lock: An object only one thread can hold at a time
  - Provides mutual exclusion

Locks

- Locks provide two atomic operations:
  - Lock.acquire() – wait until lock is free; then mark it as busy
    - After this returns, we say the calling thread holds the lock
  - Lock.release() – mark lock as free
    - Should only be called by a thread that currently holds the lock
    - After this returns, the calling thread no longer holds the lock

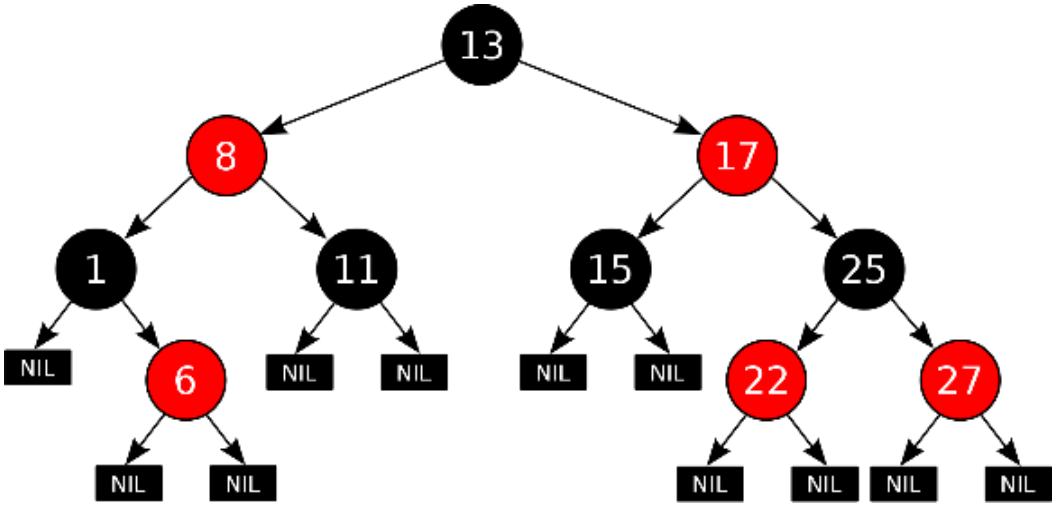
- For now, don't worry about how to implement locks!
  - We'll cover that in substantial depth later on in the class

Example: shared data structure

Thread A

Insert(3)

- lock.acquire()
- Insert 3 into the data structure
- lock.release()



Tree-Based Set Data Structure

Thread B

Insert(4)

- lock.acquire()
- Insert 4 into the data structure
- lock.release()

Get(6)

- lock.acquire()
- Check for membership
- lock.release()

OS library hooks: *pthreads*

```
int pthread_mutex_init(pthread_mutex_t *mutex,
                       const pthread_mutexattr_t *attr)

int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

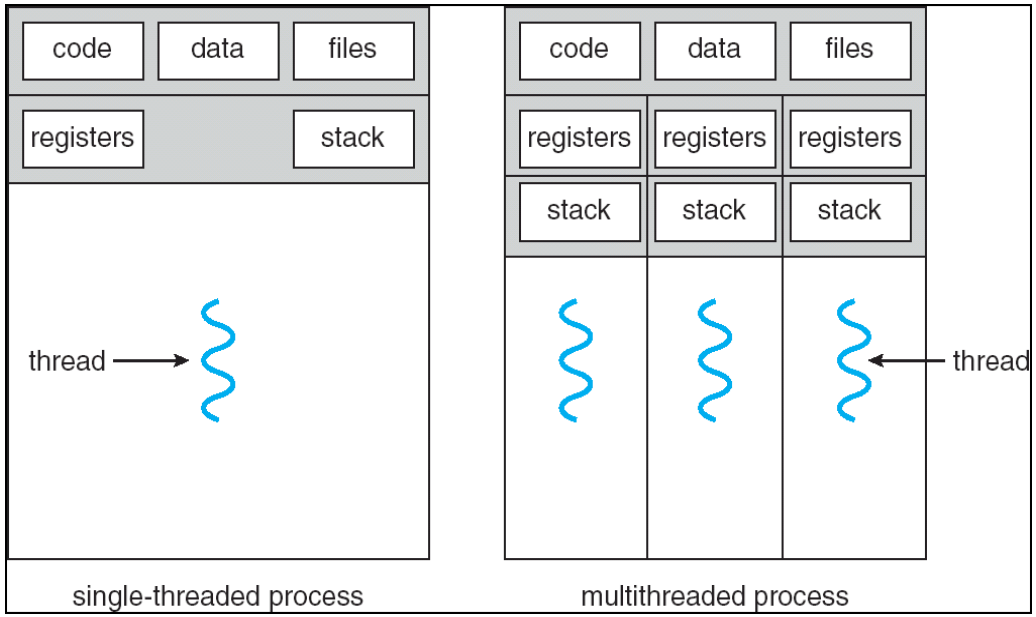
- You'll get a chance to use these in Assignment 1

Processes

Managing processes

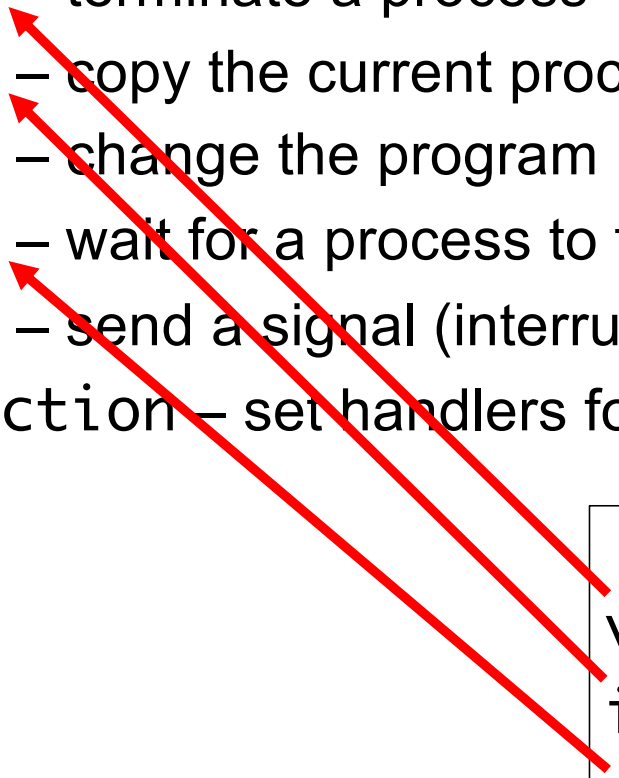
- How to manage process state?
  - How to create a process?
  - How to exit from a process?
- Remember: everything outside of the kernel is running in a process!
- Processes are created and managed by…

  processes!



Process management API

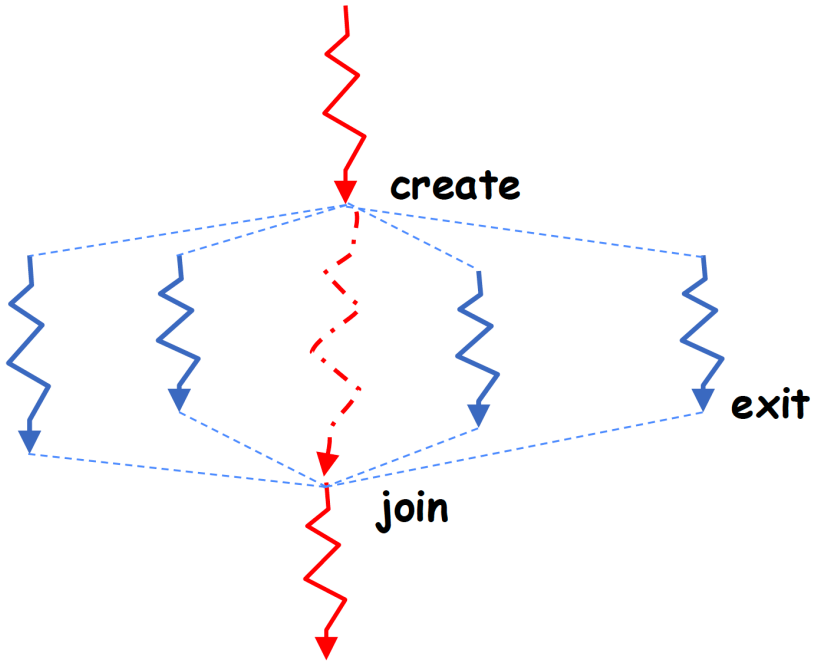
- `exit` – terminate a process
- `fork` – copy the current process
- `exec` – change the program being run by the current process
- `wait` – wait for a process to finish
- `kill` – send a signal (interrupt-like notification) to another process
- `sigaction` – set handlers for signals



Recall threads

```
void pthread_exit
int pthread_create
pthread_join
```

Recall: fork-join pattern



- Main thread *creates* (forks) collection of sub-threads passing them args to work on…
- … and then *joins* with them, collecting results.

Process management API

- `exit` – terminate a process
- `fork` – copy the current process
- `exec` – change the program being run by the current process
- `wait` – wait for a process to finish
- `kill` – send a signal (interrupt-like notification) to another process
- `sigaction` – set handlers for signals

pid.c

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>

int main(int argc, char *argv[]) {
   /* get current process' PID */
   pid_t pid = getpid();
   printf("My pid: %d\n", pid);

   exit(0);
}
```

- Q: What if we let main return without ever calling exit?
  - The OS Library calls exit() for us!
  - The entrypoint of the executable is in the OS library
  - OS library calls main
  - If main returns, OS library calls exit

Process management API

- `exit` – terminate a process
- `fork` – copy the current process
- `exec` – change the program being run by the current process
- `wait` – wait for a process to finish
- `kill` – send a signal (interrupt-like notification) to another process
- `sigaction` – set handlers for signals

Creating processes

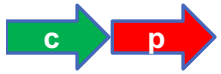
- `pid_t fork()` – copy the current process
  - New process has different pid
  - New process contains a single thread
- Return value from `fork()`: pid (like an integer)
  - When > 0:
    - Running in (original) parent process
    - return value is pid of new child
  - When = 0:
    - Running in new child process
  - When < 0:
    - Error! Must handle somehow
    - Running in original process
- State of original process duplicated in *both* parent and child!
  - Address Space (Memory), File Descriptors (covered later), etc…

fork1.c

```
#include <stdlib.h>  #include <stdio.h>
#include <unistd.h>  #include <sys/types.h>

int main(int argc, char *argv[]) {
  pid_t cpid, mypid;
  pid_t pid = getpid();                    /* get current process' PID */
  printf("Parent pid: %d\n", pid);
  cpid = fork();
  if (cpid > 0) {                          /* parent process */
    mypid = getpid();
    printf("[%d] parent of [%d]\n", mypid, cpid);
  } else if (cpid == 0) {                  /* child process */
    mypid = getpid();
    printf("[%d] child\n", mypid);
  } else {
    perror("Fork failed");
  }
}
```

fork1.c

```
#include <stdlib.h>  #include <stdio.h>
#include <unistd.h>  #include <sys/types.h>

int main(int argc, char *argv[]) {
  pid_t cpid, mypid;
  pid_t pid = getpid();                    /* get current process' PID */
  printf("Parent pid: %d\n", pid);
  cpid = fork();
  if (cpid > 0) {                          /* parent process */
    mypid = getpid();
    printf("[%d] parent of [%d]\n", mypid, cpid);
  } else if (cpid == 0) {                  /* child process */
    mypid = getpid();
    printf("[%d] child\n", mypid);
  } else {
    perror("Fork failed");
  }
}
```



fork1.c

```
#include <stdlib.h>  #include <stdio.h>
#include <unistd.h>  #include <sys/types.h>

int main(int argc, char *argv[]) {
  pid_t cpid, mypid;
  pid_t pid = getpid();                    /* get current process' PID */
  printf("Parent pid: %d\n", pid);
  cpid = fork();
  if (cpid > 0) {                          /* parent process */
    mypid = getpid();
    printf("[%d] parent of [%d]\n", mypid, cpid);
  } else if (cpid == 0) {                  /* child process */
    mypid = getpid();
    printf("[%d] child\n", mypid);
  } else {
    perror("Fork failed");
  }
}
```

p

c

Mystery: fork_race.c

```
int i;
pid_t cpid = fork();
if (cpid > 0) {
  for (i = 0; i < 10; i++) {
    printf("Parent: %d\n", i);
    // sleep(1);
  }
} else if (cpid == 0) {
  for (i = 0; i > -10; i--) {
    printf("Child: %d\n", i);
    // sleep(1);
  }
}
```

- Recall: a process consists of one or more threads executing in an address space
- Here, each process has a single thread
- These threads execute concurrently

- What does this print?
- Would adding the calls to sleep() matter?

Process management API

- `exit` – terminate a process
- `fork` – copy the current process
- `exec` – change the program being run by the current process
- `wait` – wait for a process to finish
- `kill` – send a signal (interrupt-like notification) to another process
- `sigaction` – set handlers for signals

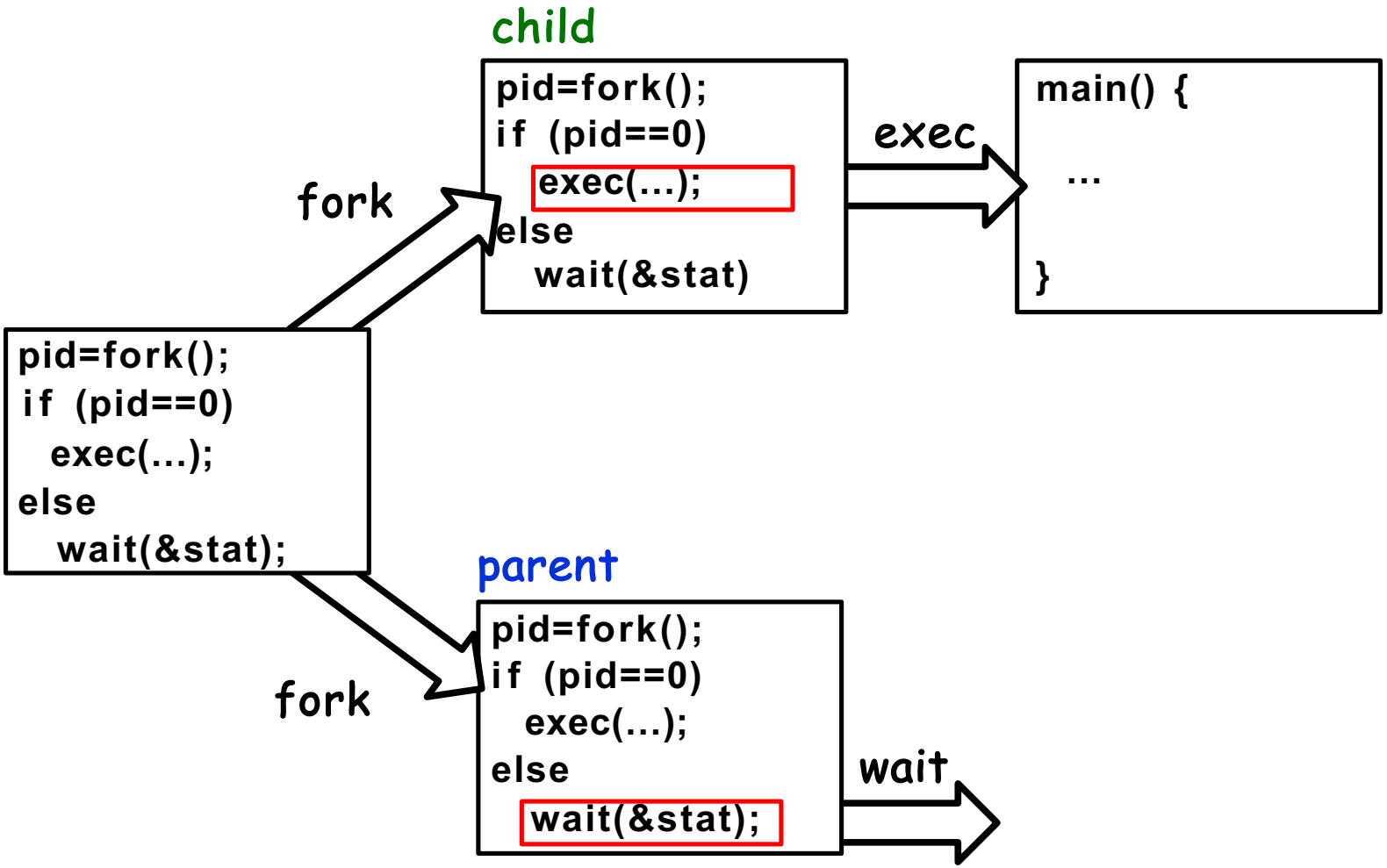
Starting a new program: variants of *exec*

```
...
cpid = fork();
if (cpid > 0) {                          /* parent process */
  tcpid = wait(&status)
  printf("[%d] parent of [%d]\n", mypid, cpid);
} else if (cpid == 0) {                  /* child process */
  char *args[] = {"ls", "-l", NULL};
  execv("/bin/ls", args);

  /* execv doesn't return when it works.
     So, if we got here, it failed! */

  perror("execv");
  exit(1);
}
...
```

fork2.c – parent waits for child to finish

```
int status;
pid_t tcpid;
...
cpid = fork();
if (cpid > 0) {                          /* parent process */
  mypid = getpid();
  printf("[%d] parent of [%d]\n", mypid, cpid);
  tcpid = wait(&status);
  printf("[%d] bye %d(%d)\n", mypid, tcpid, status);
} else if (cpid == 0) {   /* Child Process */
  mypid = getpid();
  printf("[%d] child\n", mypid);
  char *args[] = {"ls", "-l", NULL};
  execv("/bin/ls", args);
  ...
}
...
```

Process management: the Shell pattern

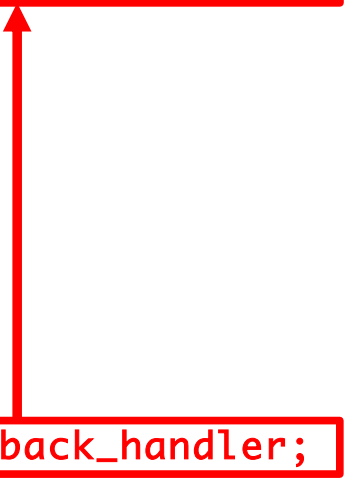


Process management API

- `exit` – terminate a process
- `fork` – copy the current process
- `exec` – change the program being run by the current process
- `wait` – wait for a process to finish
- `kill` – send a signal (interrupt-like notification) to another process
- `sigaction` – set handlers for signals

inf_loop.c

```
#include <stdlib.h>
#include <stdio.h>
#include <signal.h>

void signal_callback_handler(int signum) {
  printf("Caught signal!\n");
  exit(1);
}
int main() {
  struct sigaction sa;
  sa.sa_flags = 0;
  sigemptyset(&sa.sa_mask);
  sa.sa_handler = signal_callback_handler;
  sigaction(SIGINT, &sa, NULL);
  while (1) {}
}
```



- Q: What would happen if the process receives a SIGINT signal, but does not register a signal handler?
  - The process dies!
- For each signal, there is a default handler defined by the system

Process management API

- `SIGINT` – control-C
- `SIGTERM` – default for shell command `kill`
- `SIGSTOP` – control-Z (default action: stop process)

- `SIGKILL`, `SIGSTOP` – wait for a process to finish
  - Can't be changed with `sigaction`
  - Why?

Process vs. thread API

- Why have `fork()` and `exec()` system calls for processes, but just a `pthread_create()` function for threads?
  - Convenient to `fork` without `exec`: put code for parent and child in one executable instead of multiple
  - It will allow us to programmatically control child process' state
    - By executing code before calling `exec()` in the child
  - We'll see this in the case of File I/O later

- Windows uses `CreateProcess()` instead of `fork()`
  - Also works, but a more complicated interface

Group discussion

- Topic: processes vs. threads
  - If we have two tasks to run concurrently, do we run them in separate threads, or do we run them in separate processes?
  - What are the pros and cons?

- Discuss in groups of two to three students
  - Each group chooses a leader to summarize the discussion
  - In your group discussion, please do not dominate the discussion, and give everyone a chance to speak

(Optional) runtime-managed concurrency: illusion of threads

- Are processes and threads enough? Do we need a different design with different trade-offs?

- Languages like Golang introduces goroutines
- This adds a runtime layer between your code and OS threads
- This layer controls scheduling and switching, instead of the OS!
- Trade-off: simplicity and scale in exchange for direct OS control

- Read more here: https://omjogani.github.io/blogs/posts/the-ultimate-guide-to-concurrency-with-go-routines/

Conclusion

- System call interface is “narrow waist” between user programs and kernel
  - Must enter kernel atomically by setting PC to kernel routine at same time that CPU enters kernel mode
- Processes consist of one or more threads in an address space
  - Abstraction of the machine: execution environment for a program
  - Can use fork, exec, etc. to manage threads within a process

Thanks