CS 450: Operating Systems Lecture 4: Processes & Threads

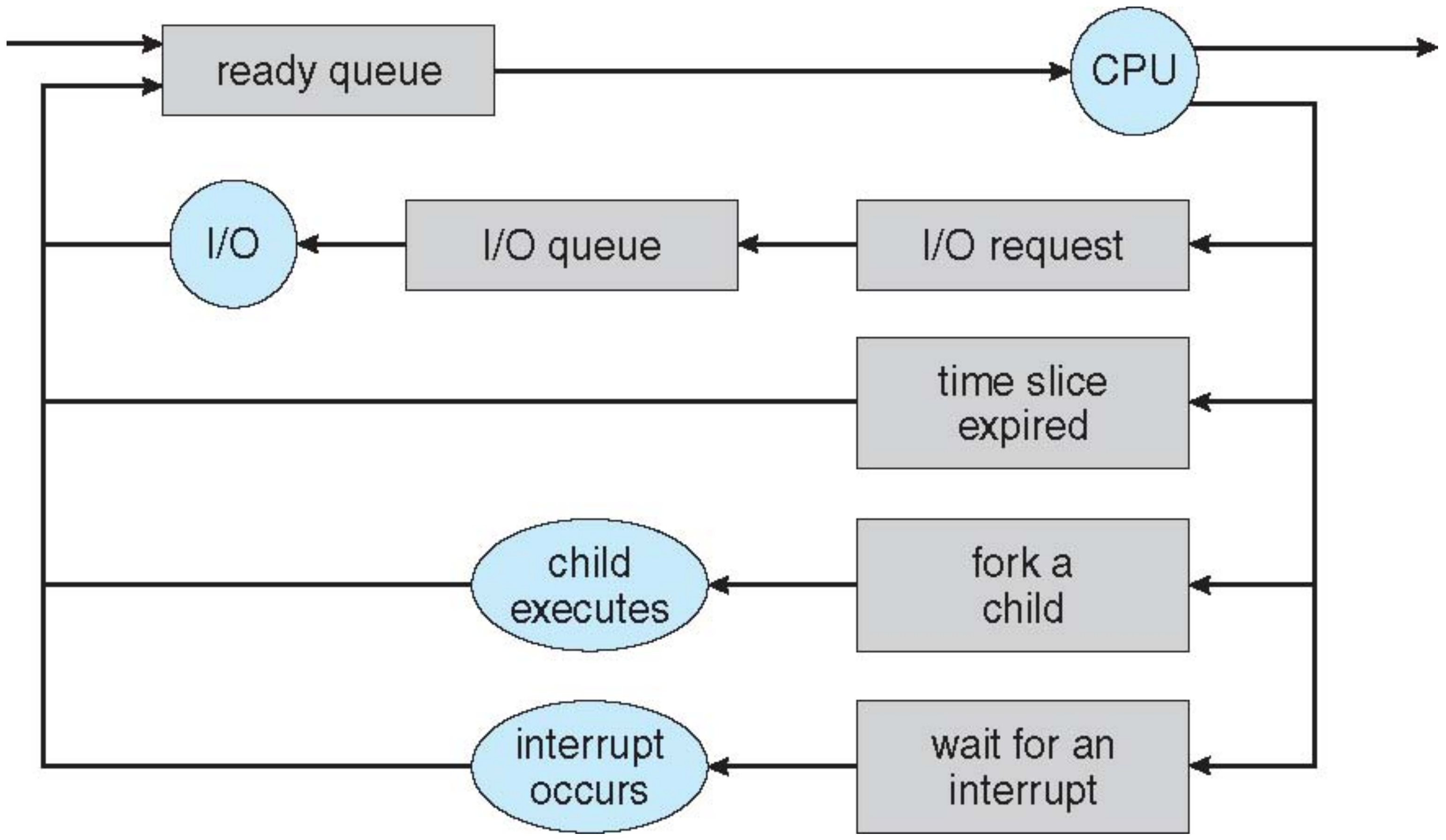
Spring 2014, J. Sasaki
Dept of Computer Science
Illinois Institute of Technology

A Touch of Scheduling

Overview of Scheduling

- We'll study scheduling in detail later.
- Multitasking OS changes between tasks to increase CPU utilization
- OS updates accounting info regularly
- Cooperative vs preemptive scheduling
- Scheduler selects a ready process to allocate CPU
  - Ready queue, Waiting queue

Process Scheduling



Schedulers

- Short-term scheduler/dispatcher quickly selects process from ready queue.
- Long-term scheduler aims for a mix of I/O-bound and CPU-bound ready processes.
- Medium-term scheduling decides which processes to swap in/out.

Thread Concepts

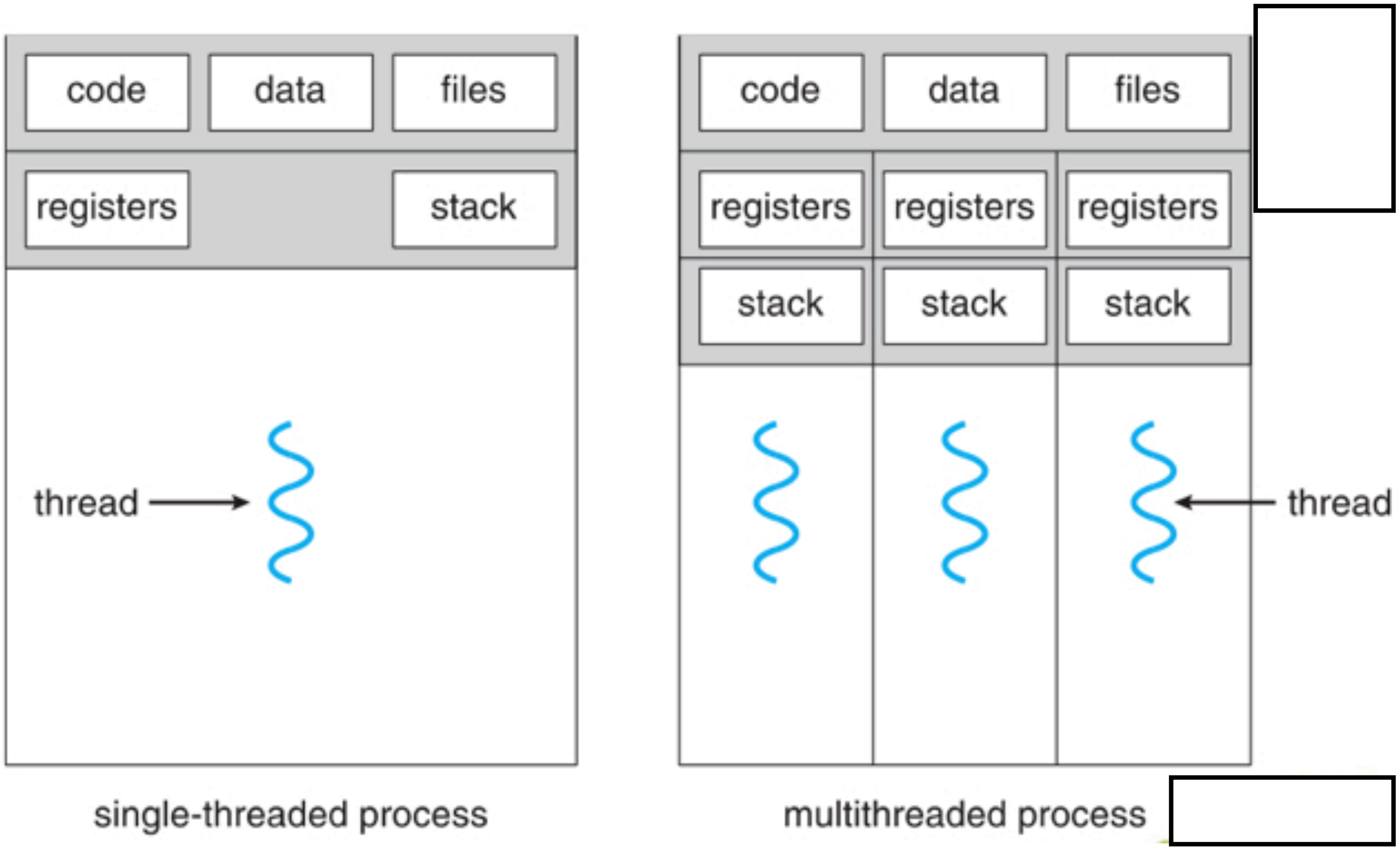
Thread

- A thread is an independent part of a program, in execution.
- Play-versus-performance view of process
  - Multiple activities on stage.
  - Each activity is a thread.
  - Share the stage, may interact.
  - Threads are individually scheduled.

Threads vs Processes

- Process has ≥ 1 thread
    - Threads work together to perform the work of the process.
    - Threads share the resources of their process — address space, files
    - Thread interaction easy, context switching faster

Multithreaded Processes



single-threaded process

multithreaded process

Thread Benefits

- Can simplify code by organizing it along multiple threads.
- Blocking a thread may still allow other parts of a process to continue working.
- Resource sharing across threads can be easier than sharing across processes.
- Threaded code may be easier to speed up with multiprocessing.

Linux Tasks

- Linux 2.4 blurred line between processes and threads.
  - Linux task that shares code, memory, open files, etc. of parent — like a thread.
  - Linux task that doesn't share — more like a process.
  - Tasks get scheduled individually by kernel.

Linux Processes

- Linux task creation heavily optimized
  - Copy-on-write memory for “unshared” address space.
- `clone()` system call used to create processes and threads
  - Process creation: `clone(`min sharing`)`
  - Thread creation: `clone(`max sharing`)`
    - (conceptually)

Threads in C

POSIX C API

- POSIX = Portable OS Interface — IEEE standards for cross-OS compatibility.
- Thread creation via `pthread_create()`
- Create a thread that executes a function.
- Wait for thread to finish: `pthread_join()`.
- Example: Lec04_thread1.c

```
#include <pthread.h>
#include <stdio.h>
void *task(void *arg);  // prototype

int main(void) {
   pthread_t thd;      // thread
   int retcode;     // 0 if thread creation succeeded

   // Create thread and have it run task
   retcode = pthread_create(&thd, NULL, task, NULL);
   printf("Thread creation returned code = %d\n", retcode);

   // Wait for thread to finish
   if (retcode == 0) {
      pthread_join(thd, NULL);
   }
}
```

```
//  Task run by thread; this one just
// prints a message.
//
void *task(void *arg) {
   printf("Thread called\n");
   return NULL;
}

/* Output:

Thread creation returned code = 0
Thread called

*/
```

Thread Argument

- 4th parameter to `pthread_create()` is a pointer to the thread task's argument.
  - Type is a general pointer: `void *`
  - Thread has to cast pointer to correct type.
- Example: Lec04_thread2.c

```
#include <pthread.h>
#include <stdio.h>

void *task(void *arg);  // prototype

int main(void) {
   pthread_t thd;       // thread
   int retcode;     // 0 if thread creation succeeded
   int thd_arg = 17;// argument to pass to thread

   // Create thread and have it run task; pass pointer to
   // thread argument.
   //
   retcode
     = pthread_create(&thd, NULL, task, (void *) &thd_arg);
   printf("Thread creation returned code = %d\n", retcode);
   printf("Thread argument at %p\n", &thd_arg);

   // Wait for thread to finish
   //
   pthread_join(thd, NULL);
}
```

```
// Thread task takes pointer to an argument value.
//
void *task(void *arg) {
   int *my_arg_ptr = (int *) arg;
   int my_arg = *my_arg_ptr;
   printf("Thread called with argument %d at %p\n",
   my_arg, arg );
   return NULL;
}

/* Output:

Thread creation returned code = 0
Thread called with argument 17 at 0x7fff5c89dac8
Thread argument at 0x7fff5c89dac8

*/

// Note: Last two lines could be swapped
```

Thread Result

- 2nd parameter to `pthread_join()` lets thread pass a result to the parent.
  - Result itself is a general pointer: `void *`
  - Pass address of the `void *` pointer
  - Parent has to cast pointer to correct type.
- Example: Lec04_thread3.c

```c
#include <pthread.h>
#include <stdio.h>

void *task(void *arg);  // prototype

int main(void) {
    pthread_t thd;    // thread
    int retcode;      // 0 if thread creation succeeded
    int thd_arg = 17;

    // Create thread and have it run task; pass
    // (pointer to) thread argument
    //
    retcode
      = pthread_create(&thd, NULL, task, (void *) &thd_arg);
    printf("Thread creation returned code = %d\n", retcode);

    int *resultptr;
    pthread_join(thd, (void **) &resultptr);
    printf("result at %p\n", resultptr);
    if (resultptr != NULL) {
        printf("result = %d\n", *resultptr);
    }
}
```

```
int nonlocal; // task will return ptr to this variable
              // task can't return ptr to local variable

// Task returns pointer to its result
//
void *task(void *arg) {
    int my_arg = *(int *) arg;
    printf("Thread called with argument %d\n", my_arg);

    nonlocal = my_arg * 2;
    printf("nonlocal at %p\n", &nonlocal);
    return &nonlocal;
}

/* Output

Thread creation returned code = 0
Thread called with argument 17
nonlocal at 0x100001078
result at 0x100001078

*/
```

Threads in Python

Python Threads

- Module `threading`, class `Thread`
  - Specify thread's code to run via constructor or override `run()` method.
  - Specify thread name in constructor, retrieve via thread.`name`
  - Call thread.`start()` to run thread; thread `is_alive()` until `run()` finishes.
  - Call another_thread.`join()` to wait until the other thread terminates.
- `threading.current_thread()` for the running thread

Lec04_thread4.py: Create threads

```
import threading
from threading import Thread
import time

def main():
   print("Start threads but don't join them\n")
   for i in range(5):
      # Create a thread that sleeps for 6-i sec
      thd = Thread(target=say_hello, \
        args=(6-i,), \
        name="mythread_" + str(i) )
      thd.start()
      print('starting {}'.format(thd.name))
      print('{} alive? {}'\
        .format(thd.name, thd.is_alive()) )
```

```
def say_hello(sleep_seconds):
    myname = threading.current_thread().name
    print('hello from {}'.format(myname))
    time.sleep(sleep_seconds)
    print('goodbye from {}'.format(myname))

main()  # run the main program
```

Sample output (Note order the threads finish)

```
hello from mythread_0
starting mythread_0
mythread_0 alive? True
hello from mythread_1
starting mythread_1
mythread_1 alive? True
hello from mythread_2
starting mythread_2
mythread_2 alive? True
hello from mythread_3
starting mythread_3
mythread_3 alive? True
hello from mythread_4
starting mythread_4
mythread_4 alive? True
goodbye from mythread_4
goodbye from mythread_3
goodbye from mythread_2
goodbye from mythread_1
goodbye from mythread_0
```

Lec04_thread5.py: Create & join threads

```
import threading
import time
from threading import Thread

def main():
   print("Start then join each thread\n")
   for i in range(5):
      # Create a thread that sleeps for 6-i sec
      thd = Thread(target=say_hello, \
        args=(6-i,), \
        name="mythread_" + str(i))
      thd.start()
      print('starting {}'.format(thd.name))
      thd.join()
```

```
def say_hello(sleep_seconds):
  myname = threading.current_thread().name
  print('hello from {}'.format(myname))
  time.sleep(sleep_seconds)
  print('goodbye from {}'.format(myname))

main() # run the main program
```

Sample output (Note order the threads finish)

```
hello from mythread_0
starting mythread_0
goodbye from mythread_0
hello from mythread_1
starting mythread_1
goodbye from mythread_1
hello from mythread_2
starting mythread_2
goodbye from mythread_2
hello from mythread_3
starting mythread_3
goodbye from mythread_3
hello from mythread_4
starting mythread_4
goodbye from mythread_4
```

Sample output:

```
hello from mythread_0
starting mythread_0
goodbye from mythread_0
hello from mythread_1
starting mythread_1
goodbye from mythread_1
hello from mythread_2
starting mythread_2
goodbye from mythread_2
hello from mythread_3
starting mythread_3
goodbye from mythread_3
hello from mythread_4
starting mythread_4
goodbye from mythread_4
```